REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE Dec 94	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE Automated testing of Applications Domains	5. FUNDING NUMBERS
6. AUTHOR(S) Richard T. Mraz	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) AFIT Students Attending: Colorado State University	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/CI/CIA 94-042D
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) DEPARTNEMT OF THE AIR FORCE AFIT/CI 2950 P STREET, BDLG 125 WRIGHT-PATTERSON AFB OH 45433-7765	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for Public Release IAW AFR 190-1 Distribution Unlimited BRIAN D. GAUTHIER, MSgt, USAF Chief Administration	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

DTIC
S ELECTE D
JAN 18 1995
B

19950117 010

14. SUBJECT TERMS	15. NUMBER OF PAGES 192
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

Memo for Record 22 Dec 94

Subject : Dissertation Submission

1. Attached are copies of my PhD abstract, bibliography, and dissertation. These are submitted according to AFITR 53-1, Paragraphs 7-7 and 7-8.

2. If you need more information regarding my dissertation, please contact me at

HQ USAFA/DFCS
2354 Fairchild Drive, Suite 6K41
U.S. Air Force Academy, CO 80841-3131

DSN: 259-3590

Richard T. Mraz
RICHARD T. MRAZ, Major, USAF
Assistant Professor, USAFA

3Atch: 1-Abstract Info
2-Abstracts (2)
3-Bibliography
4-Dissertation

Accession For	
NTIS GRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Dissertation Abstract Information

Author:	Major Richard T. Mraz
Service :	U.S. Air Force
Year :	1995
Pages :	192
Degree :	PhD Computer Science
Institution :	Colorado State University

94-042D

DISSERTATION

AUTOMATED TESTING OF APPLICATION DOMAINS

Submitted by

Richard T. Mraz

Department of Computer Science

In partial fulfillment of the requirements

for the degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Fall 1994

COLORADO STATE UNIVERSITY

December 8, 1994

We hereby recommend that the DISSERTATION AUTOMATED TESTING OF APPLICATION DOMAINS prepared under our supervision by Richard T. Mraz be accepted as fulfilling in part requirements for the degree of Doctor of Philosophy.

Committee on Graduate Work

[illegible]

Committee Member

[illegible]

Committee Member

[illegible]

Committee Member

[illegible]

Committee Member

[illegible]

Adviser

[illegible]

Department Head

ABSTRACT OF DISSERTATION

AUTOMATED TESTING OF APPLICATION DOMAINS

Test data generation is a difficult, time consuming, costly phase in the software life cycle. Software engineers address this problem by decomposing it into three phases: unit test, integration test, and system test. For each phase, testers use abstract representations of the software product to define test objectives, specify test case design strategies, and generate tests. At the system test level, we find few general purpose test data generation methods, little use of abstract representations of the system under test, and application specific test generation schemes. This research shows one way to generalize system level tests by viewing an application through its user interface. We focus on command-based systems or command language user interfaces. A test case for a command-based system is a list of fully parameterized commands. Each command in the test case is issued to the system under test and the system is examined for its response. We capture command language syntax and semantics in a domain model. The result is a test data generation method called Domain Based Testing (DBT). Testers guide test generation by defining test criteria, and map the test criteria to the domain model. The result is a test subdomain from which the test generator creates tests. To evaluate DBT and the quality of its test cases, this research uses a neural network classifier to assess test case effectiveness. The neural net classifies test case attributes/metrics into fault severity levels. Tests with low predicted effectiveness need not be run. The DBT test generation method and the neural net effectiveness prediction are applied to a command language for an industrial robot tape library.

Richard T. Mraz
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Fall 1994

ACKNOWLEDGEMENTS

PhD research requires individual creativity, inspiration, and hard work. Yet, no one earns a PhD alone. My foundation for this ordeal has been my loving wife, Mary. She kept our homefront intact, made sure our children, "remembered what Dad looked like," and was a constant source for encouragement. My children, Luke and Megan, helped me keep that balance between family and school. I thank my Mom and Dad for the comforting phone calls and letters of encouragement. My parents taught me about hard work, learning, and improving myself. That's what it takes to earn a PhD.

I cannot thank my mentor and advisor enough for her unending support. Dr. Anneliese von Mayrhauser treated me as a colleague, and I will never forget that. I could not have finished this degree without her guidance, foresight, and research talents. Thanks to Dr. Adele Howe for her guidance on our AI Planning experiments. This work helped us provide an interesting alternative to our DBT implementations. I thank my research partners and friends: Jeff Walls, Martin Shumway, Marie Vans, and Scott Gordon. They contributed to my research, my sanity, and my success. A special thanks goes to Keith Mathias. Keith has been a close friend throughout my stay at Colorado State. He helped me though some difficult times and I hope to repay my debt someday.

Finally, I must thank Mr. Robert McNitt from Storage Technology Corporation. His foresight into improving and automating his testing group was key to the success of this research. Not only did Bob provide an industrial setting to guide our work, but he also gave us the resources to continue our software tools development and test case effectiveness research.

DEDICATION

To

Mary Wheeler Mraz

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

PROBLEM STATEMENT

1.1 Introduction

Testing is one phase in the software life cycle yet it consumes "...at least half of the labor expended to produce a working program" [Bei90]. Some reasons for the cost of software testing include the price of labor (testers, test engineers, and support personnel), the cost of test environments (computers, hardware, and software), and the time required to test a software system (runtime and analysis time). For the past 30 years, research focused on reducing the time to test a software product, reducing the number of test case executions needed, and increasing test engineer productivity [Mye79, Mye76, Bei90, GH88, ABC82]. Typical goals include maximizing the yield on each test case and automating test generation and analysis.

Why is testing so costly and time consuming? Beizer and Myers answer this question by looking at the difficulty of test data generation [Bei90, Mye79]. Consider *structural testing* where the control flow graph of a program guides test case design. One measure of a thorough test set executes every path in the flow graph at least once. Unfortunately, this criteria is not practical. Even for small programs with a single loop, the number of unique paths through the program is too large to test. Another test approach views software as a "black-box" where we evaluate a program's success or failure based on its specification. One way to thoroughly test a program against its specification is to run all possible input combinations as test cases. Unfortunately, exhaustive input testing is not practical because the input space for most programs is so large we consider it infinite for testing purposes.

Because testing "all-paths" or "all-input" conditions is not feasible, testers must choose a subset of all possible tests. This subset is sometimes called the "reliable test set." If a program is correct with respect to its reliable test set, then we assume it is correct for the entire input domain. Despite its appeal, this problem is no easier than exhaustive testing. In fact, Howden shows that the definition of a reliable test set is undecidable [How87].

1.2 Solutions to the Test Data Generation Problem

Test data generation may be difficult, but computer scientists tackle complex problems all the time. They address complexity using problem solving skills like *problem decomposition* and *problem abstraction* [Boo87]. Testers use the same skills to test software. Decomposition divides a large problem into smaller, more manageable problems. Abstraction creates different views of the problem.

1.2.1 Software Testing : Decomposition

Software testing is commonly divided into three subproblems: *unit test*, *integration test*, and *system test* [Bei90, Mye79, vM90]. Unit tests are test data generation methods used on the smallest conceptual items (units) in the software system. A unit is typically a function, module, subroutine, abstract data type or object. Each unit is tested separately through a test harness or test driver. Test objectives focus on exercising the software and measuring how well "paths" through the module are exercised. Figure 1.1 shows a path selection subsumption hierarchy that relates many of the path selection criteria. We know that the highest path selection criteria all-paths is not practical. The other selection criteria in the hierarchy denote subsets of the all-paths criteria.

Integration testing concentrates on combining individual modules into a single functional unit. The functional unit can be a library, subsystem, subprogram, object, or class. Integration testing is "constructive" because modules integrate into subsystems, and subsystems integrate into higher level components. Integration tests exercise call and return interfaces between modules, evaluate the subsystem with respect to its specification, and test interaction between components. Integration testing also requires

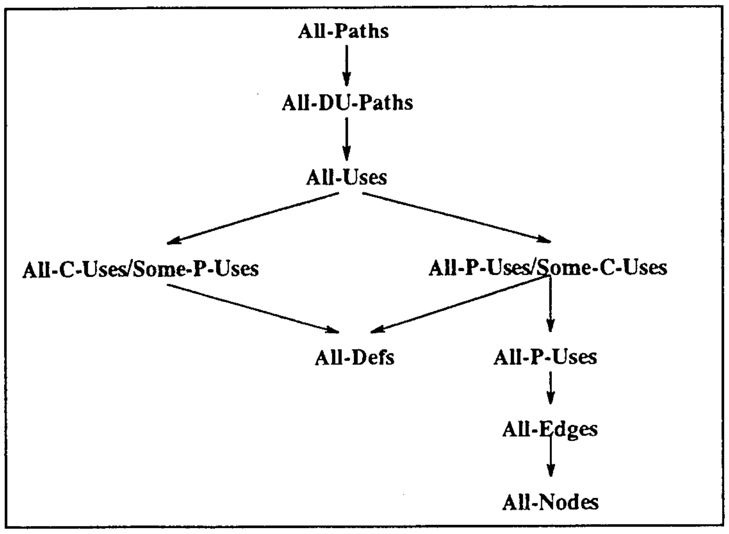


Figure 1.1: Path Selection Subsumption Hierarchy [CPRZ89]

an *integration sequence.* An integration sequence defines the order in which units are combined and tested as a subsystem. Integration sequences tend to be application dependent. A variety of integration sequencing criteria exists, e.g., top-down, bottom-up, sandwich. Typically, the most critical components are integrated first. Testers combine units into an integrated component either all-at-once or one-at-a-time. All-at-once integration combines all modules into a component at the same time. All-at-once integration is useful for small, simple subsystems, and it can be difficult to isolate faults when they occur. One-at-a-time integration is preferred for more complex subsystems. Modules are integrated one-by-one. Testers isolate a fault to the last module integrated into the subsystem.

System testing refers to testing the software product. This requires the entire application to be integrated and running. The goals for system testing are threefold. First, the system test verifies the software works as documented. Second, system test ensures the application interfaces with other systems correctly. Finally, the system test evaluates the software product against its requirements. System test also includes a qualitative assessment of the product's runtime performance, security, start-up, recovery, and configuration sensitivity [Bei90].

1.2.2 Software Testing : Abstraction

Computer scientists also use *abstraction* to address the test data generation problem. Abstraction is "the principle of ignoring those aspects of a subject that are not relevant to the current purpose in order to concentrate more fully on those that are" [CY90]. Testers use abstraction to build *representations* of the system under test. They define *test generation* methods based on these representations.

Unit testing uses two common representations: flow graphs and functional representations. Flow graphs represent the control-flow or data-flow structure of the program module. Testers may measure the quality of a test set with respect to a flow graph by traversing the graph with each test in the test set and measuring *code coverage*. Coverage measures how well the test set exercises paths in the flow graph. Flow graphs are also used in test generation methods. For example, symbolic execution uses a control-flow graph to determine execution paths, symbolically execute the path, and solve the path predicate. Functional representations view modules as individual functions. We know that exhaustive testing of all input combinations is not possible. So, test generation methods focus on reducing the number of tests required to test a module. Functional test data generation typically partitions the input-output space of each module [OB88, Bei90, Mye79]. Partitions are defined such that any value in the partition is representative of all values in the partition. If the results from a test case is correct for one value from the partition then all inputs in the equivalent class are assumed to be correct. The effect of partitioning the input-output space results in a reduced set of test cases.

Testing can also be based on specification and functional representations. Formal specification techniques include Extended Backus-Naur Form (EBNF), specification languages (e.g., Z), decision tables, and state transition diagrams [vM90]. These representations can be used to generate test cases [ROT89, OB88]. For instance, Ostrand combines a specification language with input-output space partitioning to generate test sets in a method called Category-Partition Testing. The specification language reduces the number of tests generated by partitioning the input-output space. Additional features reduce the number of tests further by defining infeasible input condition combinations and error conditions. Functional representations can be used for integration testing, too. Howden shows how to use a functional description of the integrated component and function composition for test data generation [How87]. Test data generation methods for integration testing typically partition the input-output space. These partitions focus testing on the call and return interfaces and data handling between modules.

System level test relies on formal specification or application specific representations for test data generation. The system test derives formal specification from the requirements documents and system documentation. There is little literature describing system level testing or representations used at the system level. Many system level representations are application specific. For instance, testers use formal language (i.e., grammars, attribute grammars) to automatically generate tests for parsers and compilers. [DH81, Cho77, BS82, Pur72, CB92, Pay78].

1.2.3 Test Data Generation Methods

Table 1.1 lists references to test data generation methods categorized by unit, integration, and system test. Note that most test data generation methods are at the unit level. This is not surprising because testers deal with small, manageable problems, and they use well defined representations. At the next level, we do not see many test data generation methods specifically designed for integration testing. Many of the integration testing techniques are extensions to unit test methods. For example, Category-Partition Testing can be used at the module or integration level [OB88]. Likewise, Howden shows how to extend functional testing of units to functional testing at the integration level [How87].

System testing tends to require special test data generation and application specific test data. For instance, a stress test of a system depends on qualitative system requirements, target architecture, or software version. Start-up/Recover tests are extremely application specific. For instance, each software product requires specific executable

Table 1.1: Test Data Generation Survey

Test Data Generation Method	Test Level			References
	Unit	Integration	System	
Boundary Value	X	X	X	[Mye79, Bei90, OM91]
Branch Testing	X			[Mye79, Bei90, Nta88]
Category-Partition	X	X		[OB88]
Cause-Effect Graphs	X			[OM91, vM90, Mye79, Mye76]
Data Flow	X			[CPRZ89, WO82, Nta84a, How87] [Nta84a, Wey90, FW88, Bei90] [RW82, HS89b]
Functional Testing	X	X		[How87, How86, How89] [How85, OM91]
Input Domain Testing	X	X		[WC80, Bei90, CFR90]
Mutation Testing	X	X		[DO91, CDK+89, Bei90]
Partition Testing	X			[HT90, RAO89, ROT89, WJ91, RC81]
Path Testing	X			[Mye79, Bei90, Nta88, OM91]
Random Testing	X	X	X	[Mun88, vM90, Mye79, Bei90]
Stress Testing			X	[CFR90, vM90, Bei90]
Start-up/Recovery			X	[vM90, Bei90]
Security Tests			X	[vM90, Mye79, Mye76, Bei90]
Application Specific Testing			X	[CB92, CFR90, Fis77, DH81, Het84] [LG89, Mun88, Per86, Pet85] [RAO89, vMO93, Pur72, BS82]

code segments, operating system configuration, hardware requirements, data files, and data file locations. A software product may also need system security tests for reliability assurance, data protection, secure processing, or atomic transaction processing. Test data for security testing varies widely from application to application. Consider the task of testing the security of a transaction-based banking system versus the task of testing the data protection of a companies client database. Each application needs special test data generation. We also found few *general approaches to system level test.* Instead, we found that test data generation uses application specific representations and generation tools. For instance, researchers use formal language representations to test parsers and compilers [Pur72, CB92]. Special test generators are required to stress test real-time message passing systems [Pay78].

1.2.4 Regression Testing

Testing a software product during its maintenance phase contributes additional test cost and test time. Maintenance covers software modifications, changes, and upgrades until the program is phased out. Testing software modifications is sometimes called *regression testing.* The objective is to show the software has not "regressed." Regression tests make sure old features still work, new features work as required, and modifications don't cause new problems. Tests used in the original system are one source for regression testing. Most of the time, it is economically infeasible to re-run all the test cases from the original system. Therefore, one must choose a subset of test cases that have a high potential to detect errors. One approach to regression testing is to evaluate the impact of the software changes. This impact analysis identifies the parts of the system that needs to be tested during the regression test. The goal for the regression test will be to test those system components influenced by the software change. The regression test suite will use two sources for test cases. First, we will need new test cases to test the new components. Second, we use test cases from the original version of the system. Some of the original tests can be used directly without change, others may need modification to test the software changes, and some tests can be discarded because that are no longer be applicable.

A literature review shows that few test generation methods have regression test approaches (see Table 1.2). Most research is associated with unit testing. This should be expected because most test data generation methods are at the unit level. At the integration test and system testing level, we do not find as much support for regression test methods. Regression test at the integration level extends the processes developed for unit test. For instance, [HS89b, WL92] show how to extend regression testing processes for data flow unit test into inter-procedural data flow regression testing. At the system test level, we found a single paper that relates a hierarchical representation of system requirements with software modification and regression test case selection [vMO93].

1.3 Problem Statement

The test data generation problem is complex. We can't "completely" test an application, and we do not have an algorithmic approach to choose the best subset of all possible test cases (i.e., reliable test set). Decomposing the testing problem into

Table 1.2: Regression Testing Survey

Test Data Generation Method	Regression Test Method/Process			References
	Unit	Integration	System	
Data Flow	X	X		[HS88, HS89a, HS89b, Jeo92] [vMJ93, LW91, LW89] [WL92]
Partition Testing	X			[YK87]
Path Testing	X			[BCC88]
Application Testing			X	[vMO93]

unit test, integration test, and system test attempts to address this problem. Unit and integration testing have been successful, but little research has been done on general-purpose system level test generation. This research examines automated test generation of applications at the system level. System test literature rarely describes a well defined representation for test data generation. Instead, system testing tends to be application specific using application specific representations. We intend to generalize system level tests by viewing an application through its user interface (UIF). A user interface falls into one of three categories: *command-based*, *menu-driven*, or *graphics-oriented* [vM90]. Command-based systems define a command language UIF. The interface reads a command, parses it, executes the command, and responds to the user. Menu-driven systems use a hierarchy or network of menus to read user input, execute requests, and respond to user menu selections. Graphical User Interfaces (GUI) are popular UIFs running on a variety of personal computers and workstations. GUIs use windows, icons, and pointing devices to accept user input and run the system product. Many times a GUI is layered on top of a menu-driven or command-based system [Bra92].

Of the three user interface categories, this research examines automated test generation for command-based systems. We choose command-based systems to narrow the research scope, show how to test an application through its user interface, and to generalize system level tests. One way to test a command-based system is to issue a sequence of commands and check the system for correct behavior. A *test case* is a list of fully parameterized commands from the command language that is representative of a user's session [Mos93].

Automated test generation for command-based systems requires the following:

- *Abstract Representation of the Command Language*

The key to this research is a well defined representation of the command language. The test generation method and the regression testing support rely on the command language representation. We must analyze the syntax and semantics of the command language, and we require this representation to support "most" languages.

- *Test Generation Process*

The test generation process uses the command language representation to automate test data generation. Automation relieves the tester from the low-level details of test generation, it removes test generation tedium, and it eliminates the error-prone nature of hand-generated tests. Obviously, the test generation process should follow the tester's normal work process. Otherwise, there is little chance of tool acceptance.

- *Test Case Evaluation*

The test generation process requires an evaluation for efficiency and effectiveness. Efficiency is measured in the run-time performance of the automated test generator. A fast test generation method reduces test generation time and increases testers productivity. We also require a method to evaluate test case effectiveness. Testing tends to be guided by rules-of-thumb and tester experience. We want to objectively evaluate the quality of tests generated by our method. This should be independent of the command language and the application under test.

- *Regression Test Support*

Most test data generation methods do not specify a regression test process. Our specification for the regression testing of command language modifications considers syntax and semantic changes to the language. The regression test generation method relates command language changes to the command language representation. Regression test suite definition requires rules to choose tests from the original set of test cases to make sure changes didn't break anything, and rules for generating new tests to make sure modifications and enhancements work.

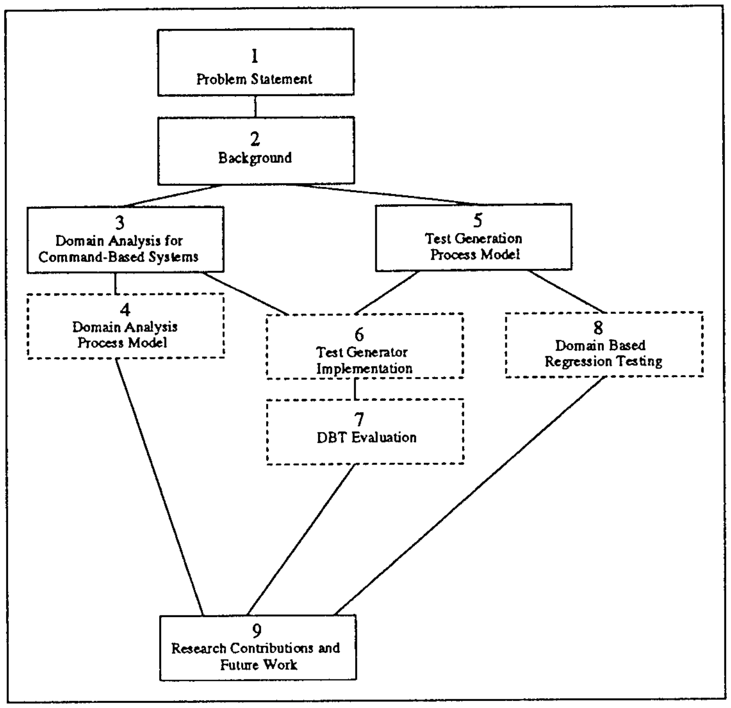


Figure 1.2: Suggested Dissertation Reading Paths

1.4 Dissertation Overview

This dissertation describes our development of an automated test generator for the system-level test of a command-based system. Chapter 2 reviews background literature required to guide the research. Chapter 3 details our analysis and representation of the command language and Chapter 4 shows how to incorporate the command language analysis into a software reverse engineering process. Chapter 5 describes the test data generation process, shows how to define test criteria, and lists several test case reuse scenarios. Chapter 6 compares two implementations of this system. Chapter 7 evaluates the test generation performance and presents an evaluation of test case effectiveness. In Chapter 8, a regression test specification is presented. This thesis can be read from cover to cover, but we realize that everyone may not need full understanding of our test generation method. Figure 1.2 suggests reading paths through the dissertation. Lined boxes denote key chapters required to understand our research. Dashed boxes represent more in-depth topics.

Chapter 2

BACKGROUND

2.1 Introduction

This chapter is organized around the four requirements defined in Chapter 1: *Representation of the Command Language*, *Test Generation Method*, *Evaluation Approach*, and *Regression Testing Support.* The first section looks at abstract representations for a command language. This is key to this research because test generation and regression testing rely on it. The second section reviews test generation, test case design strategies, and test criteria. We also examine two ways to generate tests, *grammar-based* sentence generation and *AI Planners.* The third section reviews neural network classifiers. We will use them to evaluate the effectiveness of tests generated by our test generator. Finally, we look at how to incorporate regression testing into the test generation process.

2.2 Command Language Representation

We investigated two command language representations: formal languages and domain models. Formal languages were an obvious starting point for this research. Domain models were considered because of their success in the code reuse community [Kru92, BP89, Big92]. We found them appealing for software test generation. We evaluate both representations on their ability to capture command language syntax and semantics and their usefulness in an automated test data generator.

2.2.1 Formal Languages

For the past 20 years, researchers have used formal language representation for automated test generation [Pay78, BS82, CRV+80]. They are supported by a solid theoretical foundation, they are well defined for test generation, and parser generator tools like `lex` and `yacc` can be used for automated test generation. In their most basic form, grammars represent the syntax of a language. Payne and Purdom show how to use grammars to test parsers, compilers, and real-time systems [Pur72, Pay78].

Payne extends grammar representations by adding probabilities to terminals and non-terminals of the productions [Pay78]. The probabilities alter the generation frequency of each syntactic element during test generation. For command-based systems, this feature is useful when generating tests based on an operational profile of command usage. Some researchers used the more powerful *attribute grammars* [DH81], which adds semantic actions to the productions of the language. Since we must represent command language semantics, we might consider attribute grammars. Unfortunately, these grammars are not easy to write. We also know that using attribute grammars for automated test generation becomes difficult because of the large number of semantic actions maintained by the parse tree [DH81, vM93].

This research will not use grammars exclusively. We may use grammars to represent parts of the command language, but we need a representation that does not encode command language syntax and semantics into a single representation. It is important to understand the tight coupling between the command language representation and the test generation algorithm. The test generator not only uses the formal representation of the command language but it also must consider test intent (test criteria). Therefore, we need mechanisms that let us describe test criteria (e.g., invalid command sequences, invalid parameter values, pathological tests, etc.) in the formal command language description. This requires easily adjustable descriptions and for practical purposes a set of representation mechanisms instead of a single representation.

2.2.2 Domain Modeling

Over the past ten years, software reuse has been a topic of study and empirical test [Kru92, BP89, Big92]. Historically, researchers focused on shared libraries, reusable code, and reusable programming components. Using knowledge about similar systems is a good idea from an engineering and from an economic point of view. A software engineer can build complex systems from sets of proven building blocks. The project manager can reduce project costs, time, and schedule by reusing software instead of "reinventing the wheel." Currently, software engineers are applying reuse concepts throughout the software life cycle. Successful software reuse extracts common information about a problem domain, specifies the operations of the domain, and packages the information such that one can build a new system based on the reuse knowledge. One way to capture this information is to perform a *domain analysis*, which Prieto-Diaz defines as, "a process by which information used in developing software systems is identified, captured, and organized with the purpose of making it reusable when creating new systems" [HC91]. The result of a domain analysis, a *domain model*, represents the reuse problem domain and serves as a mechanism to create instances of reusable components. Hooper summarized the importance of domain models when he stated, "Even more leverage is gained from reuse if domain analysis can derive common architectures, generic models, or specialized languages that characterize software in a special problem area" [HC91].

Neighbors coined the term *domain analysis* in his 1981 PhD dissertation [HC91]. Since then, domain analysis has been associated with the development of reusable software components [Gom91] [HC91] [Tra92] [TCY93]. Software reuse can be *horizontal* or *vertical*. Horizontal reuse refers to collections of general purpose programming tools useful across a variety of problems such as reusable data structures, sorting algorithms, and searching algorithms. Vertical reuse extracts information from a narrowly defined domain. For instance, a set of common algorithms and libraries for automated navigation calculations represents a vertical reuse domain.

Domain analysis applies to *vertical* reuse, analyzing a family of systems instead of one particular system. Domain analysis concentrates on those objects that are common in a problem domain, called "kernel objects." Optional objects or enhancements to kernel objects address the variations in the family of systems. Domain analysis commonly includes a thorough analysis of the problem, a list of domain terminology, and descriptions of the entities and operations in the problem domain. It does not require a single analysis technique. One should choose the analysis method that best fits the problem.

The result of a domain analysis is called a *domain model.* According to Gomaa, "A domain model is a problem-oriented architecture for the application domain that reflects the similarities and variations of the members of the domain" [Gom91]. Similar to domain analysis, domain models are not constrained to a single representation. Many authors suggest a representation most natural to the problem. Some of the more popular ways to represent a domain model are [BP89] [Gom91] [HC91]:

```
 1. Data Flow Diagrams
 2. Natural Language
 3. Entity Relationship Diagrams
 4. Objects
 5. Class Hierarchies
 6. Thesaurus/Classification Scheme
 7. Predicate Logic
 8. Semantic Nets
 9. Knowledge Based System
10. Predicate Logic
11. Production Rules
12. Frames
```

Booch points out, "It is impossible to capture all the subtle details of a complex software system in just one kind of diagram" [Boo91]. Therefore, more than one representation may be needed to fully specify a domain model. For instance, multiple views may be needed for different phases of the software life cycle or for different users. Multiple representations could define an abstraction hierarchy to help understand complex systems. Multiple representations can also help with modifications or extensions to the domain model. Using multiple views of the problem, one can isolate changes to the domain model as new information is learned about the problem.

Domain analysis and domain modeling look favorable for our command language representation. The domain analysis specifies how to analyze a command language for testing purposes and the domain model could capture command language syntax and

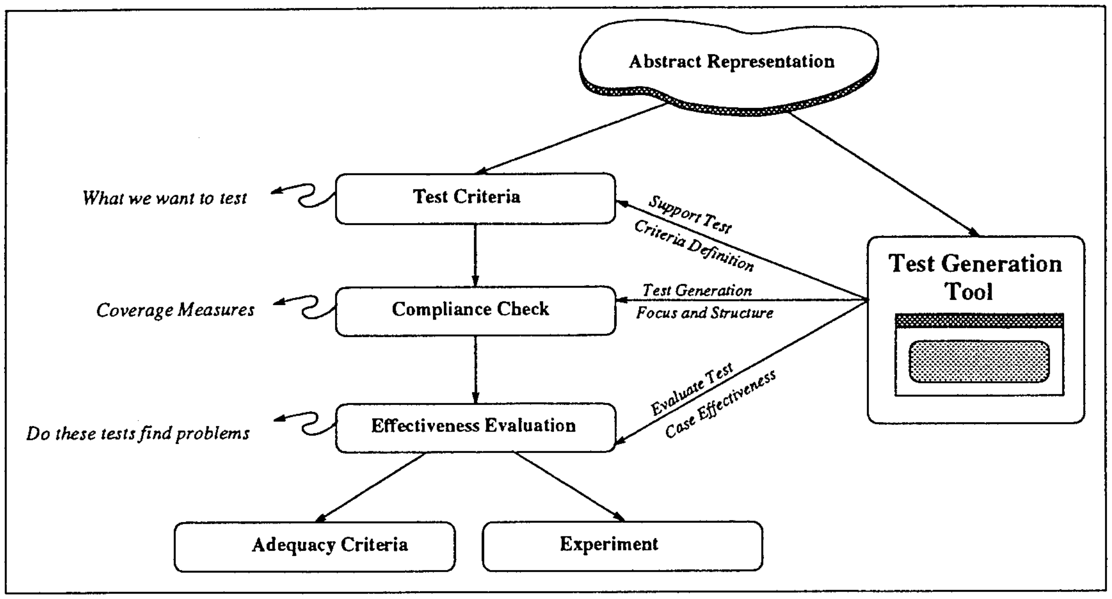


Figure 2.1: Relating Test Generation with Test Criteria, Coverage, and Effectiveness

semantics. Using multiple representations for the domain model would be helpful during test generation.

2.3 Test Generation

Automated test generation is a key part of this research so it is important to understand its relationship to the domain model representation, test criteria, and test coverage. We should also include mechanisms to evaluate test case effectiveness. Figure 2.1 shows how these factors relate to one another. The abstract representation (i.e., domain model for this research) is used as a basis for test generation. The representation is also used to represent the impact of test criteria on what is to be tested.

2.3.1 Test Criteria

A *test criteria* describes what we ought to test. Testers define a variety of test criteria for each abstract representation. For instance, common structural criteria are based on path selection such as all-statements, all-branches, and the various data flow criteria of [RW82, RW85, Nta84b, WHH80, LK83]. Test criteria imply that testing is "complete" once test data have been generated that satisfy the criteria. While testing can be complete with respect to a given test criterion, this does not necessarily imply it is satisfactory with respect to others. Relationships between a family of test criteria resolve these concerns. For instance, Clarke *et al* show how path selection criteria relate to one another using a test criteria subsumption hierarchy [CPRZ89].

2.3.2 Test Data Compliance and Test Generation

Test criteria can be used in two ways for software testing: (1) Evaluate test completeness with respect to a test criterion (resulting in coverage metrics) and (2) Test generation by providing focus and structure (generate tests that meet test criteria). Coverage metrics do not prescribe any particular test data generation approach. They evaluate how thoroughly a test set exercises the software with respect to a test criteria. Test generation uses test criteria to focus test data generation and to provide a structure to generate tests. The test criteria drive the "test generation scheme." Practically, test generators make simplifying assumptions about the system under test or about the information considered to drive test data generation. The result of such simplifications include: (1) The test data may not completely satisfy the test criteria, (2) The test data set may be larger than necessary, or (3) The test generator may be only able to generate tests for a subset of the system under test.

2.3.3 Test Case Evaluation

Test criteria and test generation based on test criteria coverage imply that high coverage defines test data that are effective, reveal faults, and isolate errors in the system under test. Yet, research shows that this is not necessarily true [HT90, TDN93]. This raises important questions for test data generation research, "How does a tester know which test criteria are best for test generation?," "Do these tests find problems, errors or faults in the software?" One way to answer these questions is to measure *test effectiveness* experimentally, analytically, or both. Axiomatic or theoretical approaches include [FW93b, FW93c, Ham89, PZ91, Wey86, Wey88, WWH91, ZG89]. Comparisons of test criteria adequacy using experimental methods include [FW93a, HT90, Nta88, Wey93a, Wey93b]. We include test evaluation as part of the research to help testers answer these questions.

In light of the relationships presented above, this research approaches the test data generation problem as follows. First, the test generator will use the domain model as a structure for test data generation. Second, *test criteria* will be defined with respect to the domain model representation of command language syntax and semantics, and the test criteria will "drive" the test generator. We will evaluate *test case effectiveness* based on *domain coverage* measures. These measures indicate how thoroughly domain model components are exercised by tests. In the following sections, we investigate test generators based on formal languages because we use formal languages as part of the domain model. We also examine the use of an AI Planner as a test generator for command-based systems. Planners use search to determine a sequence of "operations" to achieve a goal. If we encode command language commands as planner "operators," the planner could generate tests by finding a sequence of commands to reach a test data generation goal.

2.3.4 Formal Language Test Generators

Over the past 20 years, researchers used formal language theory for automated test generation. They were successful for narrow problem domains. For example, most of the early research investigated automated test generators for compilers [Pay78] [BS82] [CRV$^+$80]. Others used formal languages to generate test plans or to generate test cases for general designs and implementations [BF79] [DH81]. Efforts extending formal languages into generic test case generation were not successful because of the large number of semantic rules that must be applied.

[BS82] [BF79] demonstrate the use of formal language definitions to automatically generate test cases. Purdom [Pur72] concentrated on generating sentences from a context-free language such that each production in the grammar is used at least once. His algorithms made sure "production coverage" was met with a minimal number of sentences. In 1978, Payne developed a method to specify messages in a real-time system using a formal grammar [Pay78]. The automatic test generator created a stream of messages to "overload" test the real-time system. Message syntax was represented in BNF. Payne also associated probabilities with the terminals and non-terminals of the productions. The probabilities altered the frequencies of each syntactic unit during test generation.

Celentano *et al* [CRV+80] extended the work by Purdom to automatic sentence generation to test compilers. Using syntax-directed translation, their system could create (1) Totally incorrect, (2) Lexically correct, (3) Syntactically Correct, (4) Compile-Time Correct, and (5) Run-Time correct tests. Semantic rules for the tests were encoded in the grammar. Because test generation was based on Purdom's minimal production coverage criteria, the number of sentences in the test cases were manageable. Empirical results from testing a PL/1 compiler were successful. However, they also reported poor performance while testing an interpreter. The interpreter was defined as a finite state machine with several states and many transitions. Because the test generator tries to minimize the number of sentences, the test cases were too short and too simple to exercise the interpreter.

In a more recent paper, Duncan and Hutchison report findings from using an attribute grammar to automatically generate test cases for designs and implementations [DH81]. Their system generated test cases to compare implementations with their specification. Each test case listed inputs of the test and it defined the expected output. The system could perform structural tests, module tests, and system tests. Empirical results were shown from (1) Testing conditional statements in Ada, (2) Testing a Sort Algorithm, and (3) Testing a Text Reformatter. Unfortunately, follow up interviews with Hutchison revealed that their initial concept did not work as well as planned. During test case generation, the combinatorial explosion of semantic rules was overwhelming. Because of the many rules maintained by the parse tree, automatically generating test cases for arbitrary designs did not work. Hutchison strongly advised decomposing the test generation problem based on this experience [vM93].

From this background investigation, we know that formal languages can be used for automatic test generation. For specific problem domains, tests can be generated efficiently. For more general problem domains, the number of semantic rules makes automatic test generation impractical for reasonably sized problems. Therefore, we will use sentence generation algorithms where appropriate, but we will not use them for the entire test generation process. We will also consider the construction of "test tool generators." A test tool generator uses a description of the system under test to create a custom test data generator. We consider test tool generators to make sure our analysis is at the right level of abstraction. We do not want to construct a test data generator for a specific application or command language. Instead, we need to capture features in common to many command-based systems, incorporate them into a tool generator, and automatically create a customized test generator for each application.

2.3.5 AI Planner as a Test Case Generator

In Artificial Intelligence, planning refers to the process of generating a sequence of actions to satisfy some goal before executing the actions [CF82]. Recent uses of planning in software engineering include representation for specifications [FA88] and software reuse support [Huf92]. We will experiment with AI Planners as a test generation "engine" for this research. What makes planning an attractive method for software engineering applications is its emphasis on goals. Goal oriented sequences of actions are generated specifically to fulfill some purpose and it is easy to generate different plans for different goals. For example, in test case generation, instead of focusing on *what* commands to generate, we think about *why* we wish to test certain aspects of the system and let the planning system determine *what* actions to take.

To generate tests for command based systems, a planning system is given: (1) a description of the operators (i.e., commands from the command-based system), (2) an initial state of the world (i.e., the system being tested) and (3) a goal state (i.e., what should be tested). Operator descriptions have parameters (what objects are involved in the operator), preconditions (what must be true to use this operator) and effects

Table 2.1: Example Planning Problem [CF82]

Operator	Precondition	Effect
Pour coffee	Have brewed coffee	Problem solved
Make coffee	Have beans Have grinder Have boiling water Be in the kitchen	Have brewed coffee
Buy something	Be at store Have money	Have something
Go someplace	Place exists	Be at place Not at any other place
Get money	Be at bank	Have money
Boil water	Be in the kitchen	Have boiling water

Table 2.2: Initial State and Goal State

Start State	Goal State
Not have brewed coffee In kitchen Have grinder Have money Have boiling water	Have brewed coffee In kitchen Have grinder Have money Have boiling water

(what happens to the system after the operator executes). For our research, planner operators represent commands from the command language. Each operator description is declarative. This makes it easier to determine when operators interact, when operators must be sequenced, and how to bind parameter values between operators. Constraints on the operators are not represented explicitly, but rather defined by preconditions and effects. The initial state of the world defines the starting point for test generation. The planning system uses the initial state and operator descriptions to generate a sequence of operations that transform the initial state into a "goal" state. The goal state is the desired end-state of the system. Plans typically do not include control structures and a new plan is generated for different initial conditions or different goals.

Table 2.1 shows a list of operators, preconditions, and effects to pour a cup of coffee [CF82]. The initial state and goal states are defined in Table 2.2 [CF82]. The planning

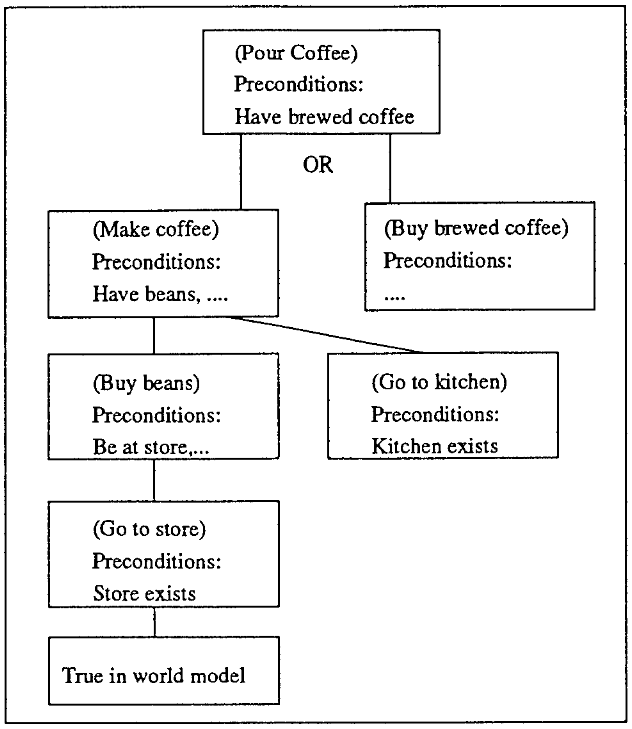


Figure 2.2: STRIPS solution to `Pour a cup of Coffee`

system looks at the difference between the current state and the goal state and applies an operator to reduce the difference. One planning system called STRIPS may create the hierarchical structure in Figure 2.2 to solve the problem. The goal is to pour a cup of coffee. The planner has two choices: (Make coffee) or (Buy brewed coffee). STRIPS chooses to (Make coffee). Some of the preconditions to (Make coffee) are satisfied in the initial state, but it must solve the precondition (Buy beans). To buy coffee beans, one must (Go to store). Since we (Have money) as part of the initial condition, we can buy the coffee beans. Because the trip to the store causes the precondition (Be in kitchen) to become False, the planner must add another step in the plan to get us back to the kitchen. The final plan is: `(Go to store) (Buy beans) (Go to Kitchen) (Make coffee) (Pour coffee)`.

Most planning systems generate a plan by in effect proving that a sequence of actions will transform the initial state into the goal state. Planning works as follows: pick a goal to achieve, find an operator whose effects include the goal, add the preconditions of the operator to the list of goals to achieve and repeat the three steps until no goals remain unresolved or all unresolved goals are satisfied by the initial state. Because the generation of the plan is based on a proof, the operator description should be complete: include all effects and preconditions of all known operators. If it is incomplete or incorrect, the plan may be as well. Rules about which operator to apply then are mostly handled by the planning system's manipulation of the operators, but may be tuned by control rules that direct selection of goals and operators.

Automated planning systems offer several potential advantages for test case generation. First, ordering the operations in the test case and checking that the order is correct is handled automatically by the planning system. Second, the representation is natural for describing commands and their interactions, information that is necessary for developing test cases. Third, the flexibility of describing new initial states and goal states makes it amenable to generating many different test cases for the same system.

2.4 Test Evaluation - Neural Network Classifiers

The test generation method requires an evaluation procedure. How good are its test cases? Do the tests identify faults? One approach relates test case effectiveness to the *domain coverage*. Domain coverage measures how well a test case exercises various components in the domain model. But, we must be careful because such measures are context dependent. They depend on the command language, the application under test, and maturity of the system under test to name a few. Therefore, we need an evaluation mechanism invariant to as many of these issues as possible. One solution is to explore the use of a neural network classifier as an effectiveness predictor. A neural classifier maps input vectors to output vectors. For test effectiveness classification, the network uses test case attributes/metrics and domain coverage as input and associates them with faults exposed by the test case. The neural net is trained to recognize this mapping for each application, command language, or software release. This opens important research topics: (1) Can we train a neural network to be an accurate effectiveness predictor? If so, we have a proof-of-concept for this approach to test case effectiveness prediction. These experiments will not only benefit this research, but may be applicable to other test case effectiveness evaluation. (2) Are domain model descriptors good for test case effectiveness prediction? The answer to this question will give us feedback on the adequacy of our test criteria. Arbitrarily defining test criteria is not a good approach to test data generation. We intend to experimentally evaluate the adequacy of our test criteria using the neural net. Information on test criteria that are likely fault indicators can be used in a feedback loop to create new tests. (3) What test case descriptors are best to train the network? This is a more difficult question. We could include all possible domain descriptors but that would make neural net training difficult and we would need a large data set for training. Our neural net evaluation should give us information about answers to this question.

Artificial Intelligence (AI) researchers developed Neural Networks (NN) to model the neural architecture and computation of the human brain [MRtPRG86]. Sometimes called "connectionist" architectures, neural nets are characterized by four features. First, a neural network consists of simple *neuron-like* processing elements. Second, processing elements are interconnected by a network of weighted connections that encode network knowledge. Third, neural networks are highly parallel and exercise distributed control. Fourth, NNs emphasize automatic learning.

Neural networks have been used as memories, pattern recall devices, pattern classifiers, and general function mapping engines [Fau94, MRtPRG86, Zur92]. Test effectiveness evaluation concentrates on their use as pattern classifiers. A classifier maps input vectors to output vectors in two phases. The network learns the input-output classification from a set of training vectors. After training, the network acts as a classifier to new vectors.

Figure 2.3 shows the anatomy of a processing element (also called a node, unit, processing unit, or neuron). Node output or activation, $o(\vec{x}, \vec{w})$, is a function of the weighted sum (dot product) of the input vector $\vec{x}$ and the interconnection weights $\vec{w}$. Figure 2.4 shows a common activation (output) function used with processing units.

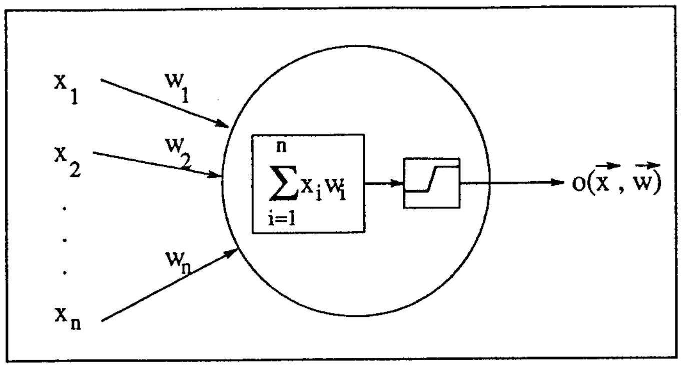


Figure 2.3: Anatomy of a Neuron

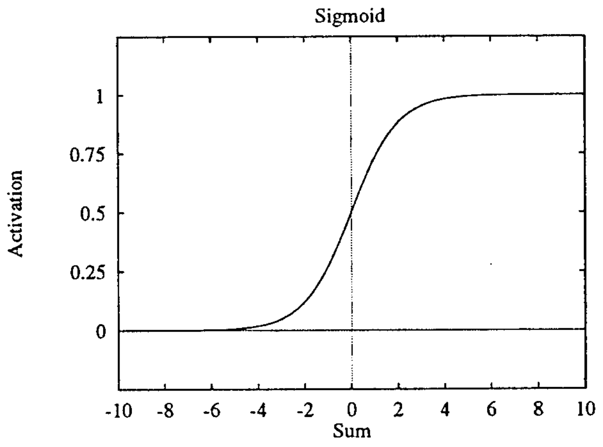


Figure 2.4: Unipolar Sigmoid Activation

Activation is a real-valued, unipolar sigmoid function. Its output is between 0.0 and 1.0 and the activation equations are:

$$sum = \sum_{i=0}^{n} x_i w_i \tag{2.1}$$

$$o(sum) = \frac{1}{1 + e^{-sum}} \tag{2.2}$$

A multilayer neural network is defined by the number of nodes in the input layer (input units), the hidden layer (hidden units), and the output layer (output units) (see Figure 2.5). The number of input units and the number of output units are defined by the classification problem. The number of hidden units is usually not known. The best way to determine the number of hidden units is through experiment. The network that

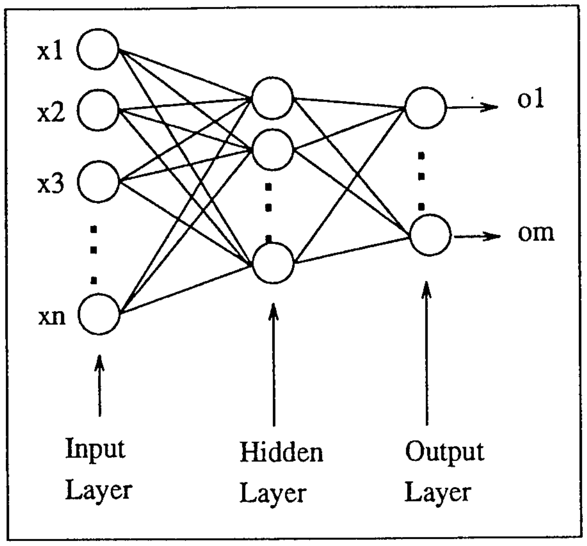
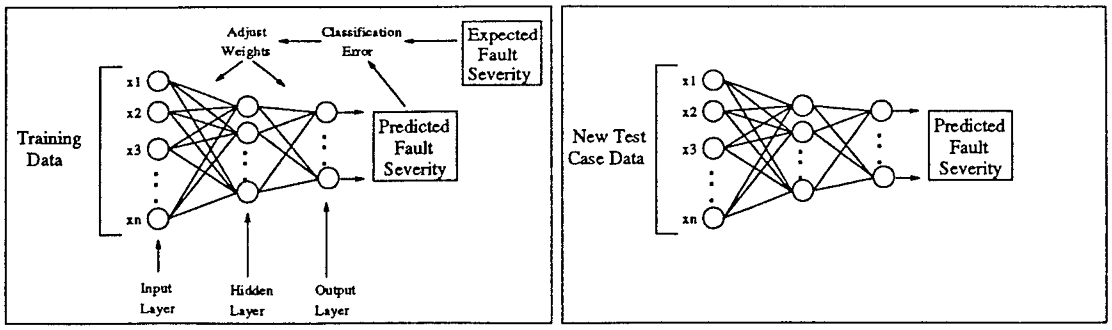


Figure 2.5: Multilayer Neural Network

produces the best classification with the fewest units is selected as the best topology. Fewer hidden units force the neural network to develop its own internal representation of the input space. Too many hidden units allow the net to "memorize" the training data instead of becoming a general purpose classifier. A few neural network training algorithms use mechanisms to automatically adjust network topology. Some start with a large topology and prune it while others start with a simple network and add hidden units or hidden layers as needed.

Backpropagation is the most popular training algorithm for multilayer neural networks. The algorithm initializes the network with a random set of weights and the network trains from a set of input-output pairs. Each pair requires a two staged learning algorithm: forward pass and backward pass. The forward pass propagates the input vector through the network until it reaches the output layer (see Figure 2.6(a)). The output of the network is compared to the expected output of the input-output pair. An error between the network output and the expected output is used in the backward pass to adjust the weights. One *epoch* is said to have passed when the network sees all input-output pairs in the training set. Training requires many epochs and stops when the sum squared error reaches an acceptable level, when a predefined number of epochs passes, or when you "give up" and conclude the network has not learned.



(a) (b)

Figure 2.6: Training Phase and Prediction Phase of the Neural Network

Training algorithms use *learning rate* and *momentum* parameters to control network weight update. Learning rate is a scaling factor that indicates how far to adjust the weights during the backward pass. Learning rates are typically set to low values [0.01...0.1]. Small learning rates result in slow learning. Large rates may move the weights too far such that the network overshoots the solution. The momentum parameter allows weight adjustment to adaptively change over the course of training. For instance, as long as the current error term and the previous error term moves the weights in the "same" direction, the weight adjustment can be larger. If the direction between the previous and current adjustment is different, then the network may be near a local minima so small weight adjustments are necessary. Typical values for the momentum term are [0.5...0.9].

Once trained, network weights are fixed and the net acts as a pattern classifier (see Figure 2.6(b)). As a classifier, the network examines input vectors it has never seen and it interpolates them into an output classification. The property of a neural network to classify patterns after training on a subset of all possible input patterns is known as *generalization*. Generalization is useful for Domain Based Testing because we cannot train the neural net on all possible test cases.

Training data is a key requirement for neural net fault prediction. Each test case must be analyzed for its attributes/metrics and we must indicate the faults exposed by the test. In the field, we use testers to perform this task, build a database of test cases, and construct training data for the neural network. The testers serve as a "test oracle" for the neural net. Sometimes it may be difficult to obtain enough data to train the neural network. High reliability systems and "rare" faults may have so few tests cases that training data may be insufficient for training or generalization.

2.5 Regression Testing

In the maintenance phase of the software life cycle, software engineers add new features, delete old functions, and fix bugs in software products. One testing strategy for software modifications is called *regression testing* [Bei90, Mye79, vMO93, Sne93]. The goal is to show the software has not "regressed." Regression tests makes sure old features still work, new features work as required, and modifications don't cause new problems. Tests used in the original system are one source for regression testing. Most of the time, it is economically infeasible to re-run all of them. Therefore, one must choose a subset of test cases that have a high potential to detect errors.

Leung and White suggested one process to create regression test suites [LW89]. Starting with the set of original test cases, each test is classified as reusable, retestable, or obsolete. A reusable test case does not test the software modification and it should produce the same results from previous tests. These do not have to be rerun. A retestable test case tests the software modification and must be rerun. An obsolete test case no longer applies to the modified software. Obsolete tests are removed from the regression test suite. Test engineers write new test cases to complete the regression test suite.

In other research, von Mayrhauser and Olender define rules for regression testing of requirement modifications [vMO93]. They represent requirements and test suites hierarchically. In the requirements hierarchy, a node represents a requirement and its children represent subrequirements. Each requirement and subrequirement has a corresponding test suite in the test suite hierarchy. An example of this structure is shown in Figure 2.7 [vMO93]. A test suite contains a set of individual test cases. Attribute vectors associated with individual requirements represent qualitative requirements. In their paper, von Mayrhauser and Olender define rules to record requirement changes in

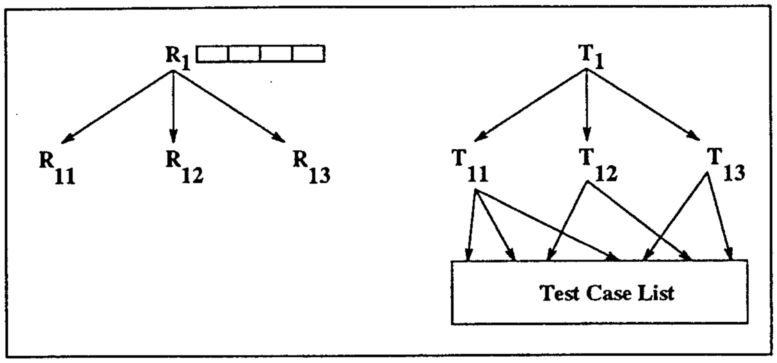


Figure 2.7: Requirements and Test Suite Hierarchies [vMO93]

the requirements hierarchy, translate requirements changes to test suite updates, and list rules for selecting a regression test suite based on the requirements modifications.

This research shows two important concepts useful to our research. First, it shows how to evaluate the original tests and partition them for regression testing. Second, it shows how to map changes in the software into the abstract representation of the problem. We will use a similar approach for regression testing a command-based system. Changes in the command languages are mapped to changes in the domain model and the changes guide rules to select tests from the original test set. We also need rules or guidelines on when new tests are required for the regression test set.

2.6 Domain Based Testing Architecture

This research examines automated test generation for command-based systems using a domain model representation of the application. We call such a testing method **Domain Based Testing** (DBT). The domain model will use a variety of mechanisms to encode command language syntax and semantics. Figure 2.8 shows the DBT architecture for this research. The architecture is designed around five subsystems: Domain Management, Test Subdomain, Test Generation, Test Evaluation, and Regression Testing. The DBT architecture figure is used throughout the dissertation as a guide for each chapter.

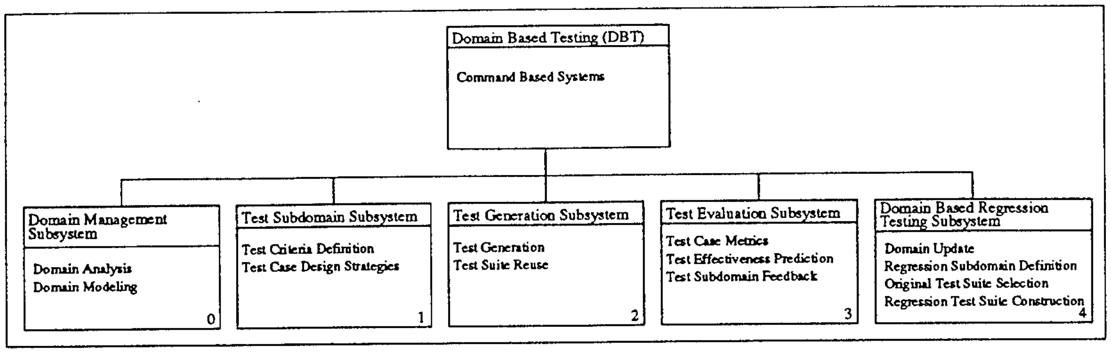

Domain Based Testing (DBT)
Command Based Systems
Domain Management Subsystem
Domain Analysis
Domain Modeling
0
Test Subdomain Subsystem
Test Criteria Definition
Test Case Design Strategies
1
Test Generation Subsystem
Test Generation
Test Suite Reuse
2
Test Evaluation Subsystem
Test Case Metrics
Test Effectiveness Prediction
Test Subdomain Feedback
3
Domain Based Regression Testing Subsystem
Domain Update
Regression Subdomain Definition
Original Test Suite Selection
Regression Test Suite Construction
4

Figure 2.8: Domain Based Testing Top Level Abstract Machine Diagram

2.6.1 Domain Management Subsystem

Domain Based Testing is a test generation method based on domain analysis and domain modeling. Both of these techniques became popular with code reuse researchers. We use them to model a system under test, as a basis for test generation and as a structure to generate regression tests. The Domain Management Subsystem (DMS) includes tools, utilities, and editors to capture the domain model. Chapter 3 defines domain analysis steps and Chapter 4 shows how to incorporate the analysis into a software reverse engineering process.

2.6.2 Test Subdomain Subsystem

The *Test Subdomain Subsystem* couples software testing strategies with the domain model representation. The tester defines the test criteria by altering, modifying, and configuring the domain model. Any modification to a domain model is called a *test subdomain.* Chapter 5 shows how to map a test criteria into domain model modifications and describes two example test subdomains.

2.6.3 Test Generation Subsystem

The goal of DBT is to automate test generation of application domains. The *Test Generation Subsystem* uses information from the domain model, the test subdomain, and the test engineer to generate test suites. Chapter 5 combines all three to generate test cases for a command based system. Chapter 6 describes two test generation implementations for DBT. The first uses a hybrid collection of sentence generation algorithms and utilities and the second uses an AI Planner.

2.6.4 Test Evaluation Subsystem

This subsystem evaluates the efficiency and effectiveness of DBT. Efficiency is measured by a run-time evaluation. Run time is important for this test generation method because it is an automated, interactive test generator. The effectiveness evaluation uses a neural network to predict test case effectiveness. Metrics, coverage measures, and test case attributes are used as input to a neural classifier. The network learns to map these measures into fault severity levels. Once the network is trained, it acts as a fault effectiveness predictor for new tests. Chapter 7 details the results of the DBT evaluation.

2.6.5 Domain Based Regression Testing Subsystem

The fifth subsystem, *Domain Based Regression Testing* (DBRT), defines one way to construct regression test suites based on a domain model representation of a command language. This includes update rules for the domain model, rules to select tests from the original test suites, and a definition of a regression subdomain for generating tests that test the domain modifications. We explain the concepts and describe the details of DBRT in Chapter 8.

Chapter 3

DOMAIN ANALYSIS FOR COMMAND BASED SYSTEMS

3.1 Introduction

This chapter presents the domain analysis for command-based system testing. The resulting domain model is used as the representation for our test generation method. We define the analysis steps sequentially, but they should be used iteratively to refine the domain model. This chapter defines the steps used by the Domain Management Subsystem in the Domain Based Testing Architecture (see Subsystem 0 in Figure 3.1). We use Input-Process-Output (IPO) diagrams to specify each step and we use a command language for an industrial tape library for illustration throughout the chapter.

3.2 Example Problem Domain: Robot Tape Library

Storage Technology Corporation (StorageTek) produces an Automated Cartridge System (ACS) that stores, reads, writes, and retrieves magnetic cartridge tapes [Tek92]. The system maintains cartridges in a 12-sided "silo" called a Library Storage Module

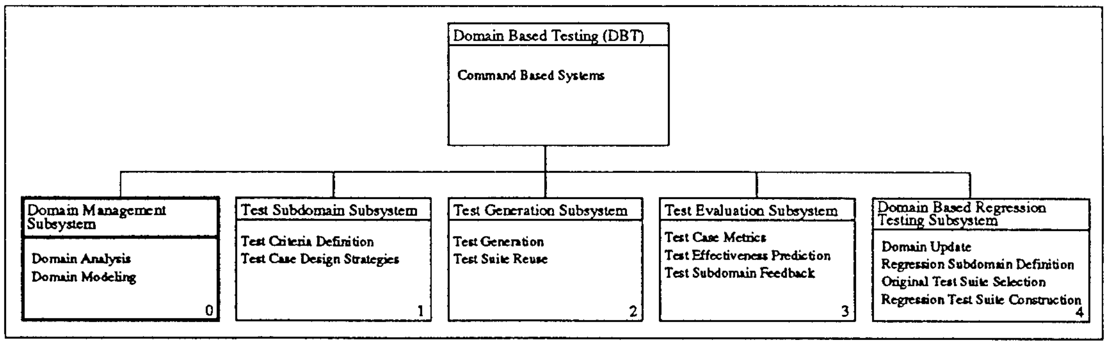

Domain Based Testing (DBT)
Command Based Systems
Domain Management Subsystem
Domain Analysis
Domain Modeling
0
Test Subdomain Subsystem
Test Criteria Definition
Test Case Design Strategies
1
Test Generation Subsystem
Test Generation
Test Suite Reuse
2
Test Evaluation Subsystem
Test Case Metrics
Test Effectiveness Prediction
Test Subdomain Feedback
3
Domain Based Regression Testing Subsystem
Domain Update
Regression Subdomain Definition
Original Test Suite Selection
Regression Test Suite Construction
4

Figure 3.1: Domain Based Testing Top Level Abstract Machine Diagram

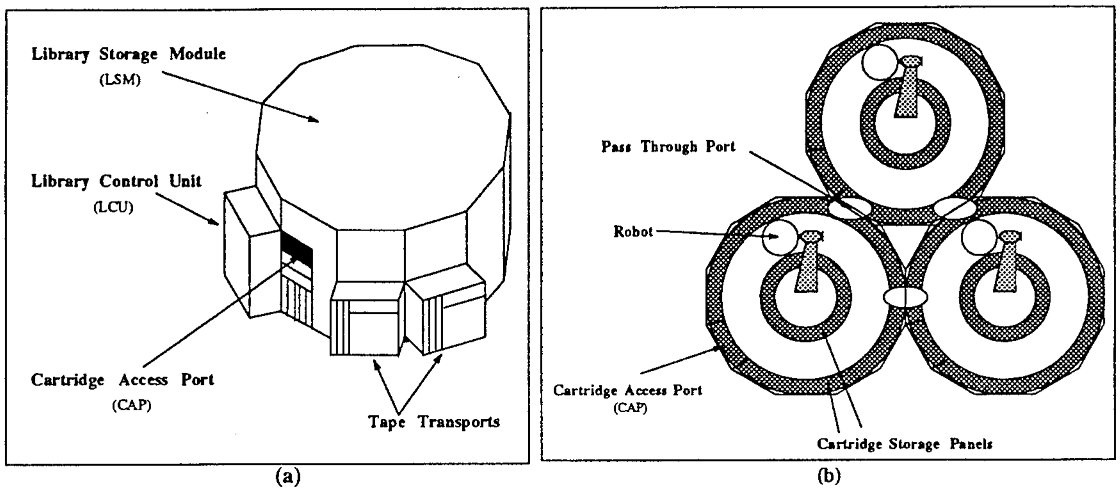


Figure 3.2: Automated Cartridge System with Three LSMs [Tek92]

(LSM). Each LSM contains a vision-assisted robot and storage for up to 6000 tapes. Tapes occupy cells in the outer and inner panels. New tapes can be entered through a special door called a Cartridge Access Port (CAP). Figure 3.2(a) shows a single LSM with tape drives, access port, and control unit. The robot inside the LSM identifies tapes using an optical scanner. Once a tape is identified, the robot can move the tape to a cell, mount the tape in a tape drive, dismount tapes, or eject tapes through a CAP. One ACS can support up to sixteen LSMs. Figure 3.2(b) shows a top-down look at an ACS with three LSMs. Tapes move between LSMs through special doors called "pass-through-ports." The ACS and its components are controlled through a command language interface called the Host Software Component (HSC). Each HSC supports from one to sixteen ACS systems. HSC commands manipulate cartridges, set the status of various components in the system, and display status information to the operator's console. The command language consists of 30 commands and 45 parameters. Appendix 1 lists the domain model for HSC Release 1.2 and we use parts of its description to illustrate domain analysis steps throughout this chapter.

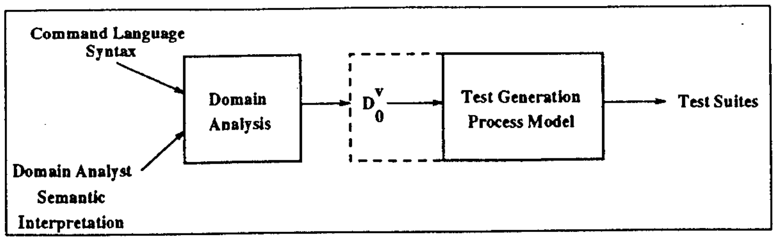

Command Language Syntax
Domain Analysis
Domain Analyst Semantic Interpretation
D_0^v
Test Generation Process Model
Test Suites

Figure 3.3: Top Level Domain Analysis IPO Diagram

3.3 Domain Analysis Overview

The Input-Process-Output diagram in Figure 3.3 shows the top-level view of Domain Based Testing. Domain analysis produces the domain model and the test generation process uses the domain model to create test suites. Input to domain analysis includes syntactic information about the application and semantic interpretation from the domain analyst. The analyst supplies this information based on knowledge of the application domain, user documentation, and manuals. The results of the domain analysis produce a domain model, D_0^v. The zero subscript identifies the domain model as the initial domain from which tests are generated and the superscript denotes the version of the system under test. The figure also shows how the domain model is used-by and is an integral part-of the Test Generation Process Model. The output of the test generation process are test suites.

Table 3.1 lists domain analysis steps used to analyze a command language. We list the steps sequentially but one should apply them iteratively to refine domain model definition. The resulting domain model has six components, *Object Definition*, *Object Hierarchy*, *Command Syntax Definition*, *Command Semantic Rules*, *Script Class Definition*, and *Script Rule Definition.*

3.4 Command Language Analysis

DBT requires certain properties of a command language, its syntax and semantics. The syntax of the command language must map to objects and behaviors of the system under test. Commands (actions on objects) and command parameters must abstract to

Table 3.1: Domain Analysis Steps for Domain Based Testing

Domain Analysis Step	Domain Model Component
1. Command Language Analysis	
1.1. Identify/Define a Command Language Interface	
1.2. Check Command Language to Object Mapping	
1.3. Create Command Language Glossary	Command Glossary
2. Object Analysis	
2.1. Define Objects and Object Elements	Set of Objects
2.2. Define Default Parameter Values and Create Object/Object Element Glossaries	Default Parameter Sets
2.3. Define Object Hierarchy	Object Hierarchy
2.4. Annotate Hierarchy with Parameter Constraints	Parameter Constraint Rules
3. Command Definition	
3.1. Syntax Representation	Command Language Syntax
3.2. Identify Pre/Post Conditions	Updated Command Glossary
3.3. Identify Intracommand Rules	Intracommand Rules
4. Script Definition (Command Sequencing)	
4.1. Script Class Definition	Script Classes
4.2. Script Rule Definition	Command Sequencing Rules

objects in the application. Without a good mapping between the parameters and objects of the system, we may not have enough information to properly model the application semantically or syntactically. An example demonstrates the need for defining minimum command language properties. Suppose we must test a C compiler. The proper level of testing is not the command line compiler invocation (e.g., `cc file.c`), but the *content* of the source file (`file.c`). The domain of the C compiler is the C language. The testable entities are C source programs (i.e., the contents of the source file). In contrast, the StorageTek HSC command language directly describes the operation of the robot tape library. It contains the properties required for DBT. If the command language meets DBT syntax and semantic properties, we record each command in a *command glossary* keyed by command name. We enter a short description of the command and the command's syntax. Table 3.2 shows a entry for the HSC `DISMOUNT` command.

3.5 Object Analysis

Step two in the DBT domain analysis is to identify the *objects* of the system, *object elements*, and *object relationships*. This analysis captures the entities testable from

Table 3.2: Command Glossary Entry for the DISMOUNT command

Command Name	DISMOUNT
Syntax	Dismount-Cmd ::= DISMount { , | <volser>} <drive-id> [<host-id>]
Description	Dismount a cartridge tape from a tape drive

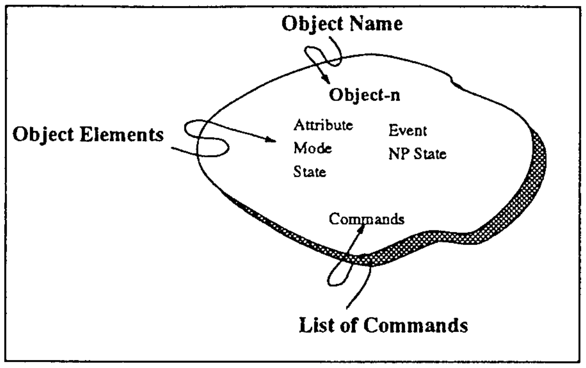


Figure 3.4: Anatomy of an Object

the command language. Test generation uses this information during parameter value selection and for parameter constraint rules. *Objects* denote physical or logical entities from the problem domain [Boo83, Boo91, Mul89]. We use command language syntax, user documentation, and analyst interpretation for object identification. Command language parameters map to objects of the system and we categorize them as *object elements*. Figure 3.4 shows a generic object used in DBT domain modeling diagrams. Each object has a name, object elements, and a set of commands. In the next sections, we define each part of the object in more detail.

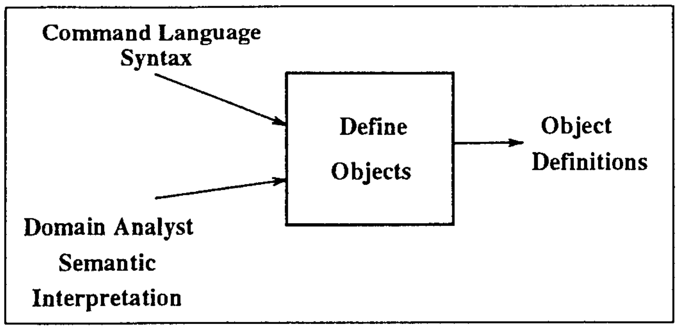


Figure 3.5: IPO Diagram : Object and Object Element Definition

3.5.1 Object and Object Element Definition

Figure 3.5 shows the IPO diagram used to identify objects in the problem domain. One input to the process is syntax from the command language. The domain analyst interprets their semantics with the use of user documentation and knowledge of the application. The output is a set of object definitions and the process uses the following steps:

1. Analyze command parameters for problem domain entities.
2. Define an object for each logical and physical entity.
3. Associate parameters (object elements) with an appropriate object.
4. Classify object elements by type.
5. Define possible values for each object element.

Each parameter in the command language is categorized according to the object it influences. This classification provides a first cut of the objects and their object elements in terms of command language parameters. Figure 3.6 shows how parameters from two HSC commands relate to three domain objects: **Cartridge, Tape Transport,** and **HSC**. Analyzing the remaining commands reveals the objects listed in Table 3.3.

This object analysis provides an initial list of object elements. Because the analysis is based on command syntax, it naturally contains information directly controllable by the command language. Depending on the command language, there may be other attributes associated with objects in the application domain that are relevant to test data generation. Thus the parameters associated with each object are a subset of

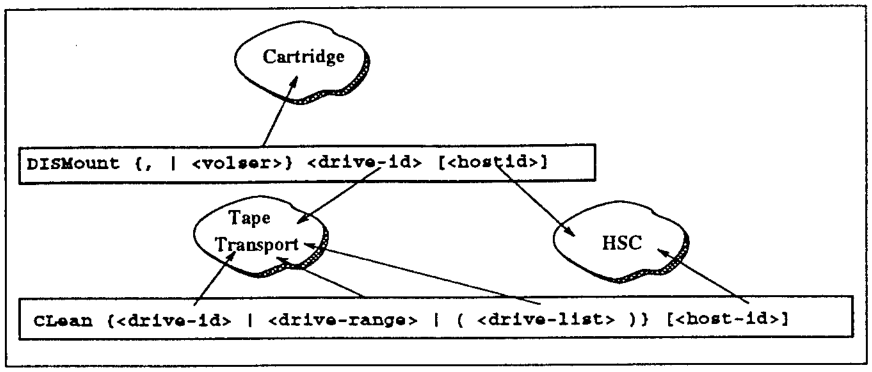

Cartridge
DISMount {, | <volser>} <drive-id> [<hostid>]
Tape Transport
HSC
CLean {<drive-id> | <drive-range> | (<drive-list>)} [<host-id>]

Figure 3.6: Analyzing HSC Commands for Objects and Object Elements

Table 3.3: Objects in the Robot Tape Library Domain

Object	Abbreviation	Description
Host Software Component	HSC	Operating system software used to control the robot tape library.
Documentation		On line documentation.
Console		Operator's console identifier.
Automated Cartridge System	ACS	A collection of one or more LSMs.
Scratch Pool		Set of scratch cartridges.
Library Management Unit	LMU	Commands robot.
Library Storage Module	LSM	A single "silo" where cartridges are stored.
Cartridge		Storage medium.
Control Data Set	CDS	Contains volume information about all cartridges.
Playground		Reserved area for cartridges during LSM initialization.
Pass Through Port	PTP	Access door between LSMs.
Tape Transport		Tape drive that reads/writes cartridges.
Panel		Racks located inside an LSM. Used for cartridge storage.
Cartridge Access Port	CAP	A special door to enter and retrieve cartridges.
Pass Through Port Column		A column of cartridge locations on the PTP.
Row		A row of cartridge locations on a panel.
Column		A column of cartridge locations on a panel.
CAP Row		A row of cartridge locations on a CAP.
CAP Column		A column of cartridge locations on a CAP.

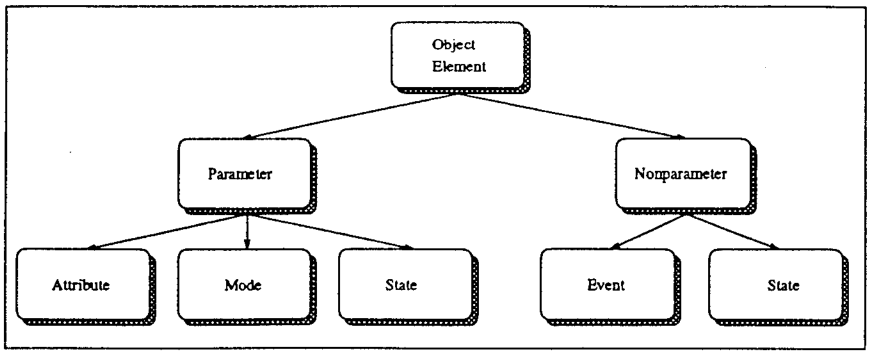


Figure 3.7: Object Element Type Classification

object elements. Object elements are similar to the concept of *object attributes* in OOA/OOD [Boo83, Boo91, RG92]. Attributes define qualities and properties of the object. Attributes may place constraints on an object such as limiting the range of possible values, forcing the selection of a particular value, or indicating dynamic behavior of the object. The difference between object attributes and object elements is found in the information that is described. For testing, we do not need as much information about an object when compared to the amount of information needed to instantiate and implement one. While we use the object model to generate test cases, we do not have to instantiate every object in the system to generate meaningful tests.

Object element classification places object elements into one of five mutually exclusive categories as shown by the leaf nodes in Figure 3.7. The first classification partitions object elements into those that are part of the command language (*parameter*) and those that are not (*non-parameters*). Object elements related to command language parameters can be *parameter attributes*, *mode parameters*, or *state parameters.* Sometimes domain analysts provide semantic information that cannot be found in the parameters of the command language. These object elements are called *nonparameters.* The following subsections describe each object element type.

Parameter Attribute: Once classified as an object element, a parameter of the command language becomes an *attribute* when it uniquely identifies instances of objects. For example, the *lsm-id* attribute uniquely identifies a particular Library Storage Module (LSM). Each object may have one or more parameter attributes, although in our

Table 3.4: Parameter Attributes in the Robot Tape Library Domain

Object	Attribute
Host Software Component	host-id
Documentation	msg-id
Console	console-id
Automated Cartridge System	acs-id
Scratch Pool	subpool-name
Library Management Unit	station
Library Storage Module	lsm-id
Cartridge	volser
Control Data Set	dsn
Playground	playgnd-cc
Pass Through Port	ptp-id
Tape Transport	drive
Panel	pp
Cartridge Access Port	cap-id
Pass Through Port Column	ptp-cc
Row	rr
Column	cc
CAP Row	cap-rr
CAP Column	cap-cc

analysis most objects have only one. Table 3.4 lists the parameter attributes for all HSC objects.

Mode Parameter: A *mode* is a parameter of the command language that sets an operating mode or a warning mode for an object. An operating mode defines how the system behaves when an error occurs or when an event takes place. A warning mode can identify the type of warning messages, where the warnings appear, or the conditions for issuing warnings. Because they appear as parameters of the command language, operating modes can be changed by issuing the appropriate command. Table 3.5 lists mode parameters for HSC objects.

State Parameter: A *state parameter* is a parameter of the command language that sets the state of the object. Because *state parameters* are part of the command language, one can change their value by issuing a command with an appropriate parameter value. Parameter state is important semantic information for test case generation. These will later be incorporated into the next analysis step as preconditions and postconditions for commands. For instance, the system may need to be in a particular state before issuing a command (i.e., a precondition). If the object is not in the proper state, then

Table 3.5: Mode Parameters in the Robot Tape Library Domain

Object	Mode
Host Software Component	baltol comp-name deferred dismount entdup float full-journal inittime initwarn maxclean mount-msg output scratch sectime secwarn viewtime vol-watch
Automated Cartridge System	acs-scr-threshold acs-subpool-threshold
Scratch Pool	subpool-threshold
Library Storage Module	lsm-scr-threshold lsm-subpool-threshold

a command can be issued to ensure semantic correctness by changing the system state. For example, the `SRVLEV` command for HSC sets the service mode to `Base` or `Full`. Full service is required for normal operations. Many commands cannot execute successfully at a `Base` mode. Table 3.6 lists state parameters for HSC objects.

Nonparameter Event: Some object elements are not part of the command language. Domain analysts identify these elements from their knowledge of the semantics of the

Table 3.6: State Parameters in the Robot Tape Library Domain

Object	State
Host Software Component	autoclean gdg-sep separation service-level specvol zeroscr
Cartridge Access Port	prefvlu
Library Management Unit	lmu-status
Library Storage Module	lsm-status

system under test. While they cannot be controlled by the command language, they may be important for test case generation. Particularly, events that happen as a consequence of system operation may influence test case generation after the event occurs. For instance, attempting to `Enter` another cartridge tape into a full LSM is not possible. The *event* "lsm-full" is not controllable by the command language. Nonparameter object elements associated with events are called *nonparameter event.*

Nonparameter State: *Nonparameter State* elements do not appear as parameters of the command language. They represent object state that cannot be set through a parameter choice. They are results of the side-effects of executing a command or a sequence of commands. It may not be possible to change the value of a nonparameter state element on demand. For example, in the robot tape library, the Cartridge Access Port (CAP) is a special door used to enter and eject tapes from a silo. Using the `ENTER` command, tapes can be placed in the door and the robot will move them into the silo. Once an `ENTER` command is issued to a particular CAP, the CAP cannot be used until released by the `DRAIN` command. The "state" of the CAP can also be changed as a side-effect of the `ENTER`, `EJECT`, `SENTER`, `DRAIN`, and `RELEASE` commands.

3.5.2 Default Values and Glossaries

At this point in the domain analysis, we have defined objects of the system, associated object elements with each object, and classified each object element by type. Now, in Step #2.2, we will define default parameter sets for each object element and create two glossaries. The first is called the *object glossary* and the second is the *object element glossary.* The object glossary maintains information about each object by recording its name, a short description, a list of commands associated with the object, and the names of its object elements. Table 3.7 shows the **LSM** entry from the Object Glossary.

The *object element glossary* stores detailed information about each object element keyed by object element name. The glossary lists the range of values for each element, the representation of each object element, and the default set of values for each object element. This information is needed for automated parameter value selection during

Table 3.7: Object Glossary Entry for the **LSM** Object

Object	**LSM**
Description	Library Storage Module - A single tape "silo"
Commands	`DISPLAY MODify MOVE VIew Warn`
Parameter Attribute	*lsm-id*
Mode Parameter	*lsm-subpool-threshold* *lsm-scr-threshold*
State Parameter	*lsm-status*
Nonparameter Event	*lsm-full*
Nonparameter State	

test case generation. Table 3.8 shows several entries from the Object Element Glossary for the StorageTek HSC command language.

3.5.3 Define Object Hierarchy

We now have represented the system under test as a collection of objects with detailed information about their object elements. The next step in the domain analysis is to show relationships between the objects. These relationships are captured in an *object hierarchy.* The IPO diagram to define the object hierarchy is shown in Figure 3.8. Input to the process includes the collection of objects from the previous step. With the help of the analyst's semantic interpretation we build a structural object hierarchy. Object relationships are defined as a structural hierarchy because we want to capture the "part-of" relationships between objects (i.e., object1 is part-of object2). The StorageTek domain provides a good example for constructing an object hierarchy (see Figure 3.9). Consider the ACS object. Each ACS supports up to sixteen LSMs, and this relationship is shown in the figure as an arrow from the ACS object to the LSM object. Each LSM contains panels, tape drives, cartridge access ports, etc. Arrows from the LSM to each object denote this structure.

3.5.4 Annotate Hierarchy with Parameter Constraints

In Step #2.4 of the domain analysis (Table 3.1), we annotate the object hierarchy with semantic rules about parameter values. The IPO diagram for this step is shown in Figure 3.10. The input to the process includes the object hierarchy from the previous

Table 3.8: Entries from the HSC Object Element Glossary

Parameter Name		
lsm-id	Full Name	Library Storage Module (LSM) Identifier
	Definition	Names an Instance of an LSM within an ACS
	Type	parameter attribute
	Values	000...FFF
	Object	LSM
	Representation	Range
maxclean	Definition	Number of times a cleaning cartridge is used before ejecting
	Type	mode parameter
	Values	10...100
	Object	HSC
	Representation	Range
lmu-status	Definition	Status of the Library Management Unit (LMU)
	Type	state parameter
	Values	UP | DOWN
	Object	LMU
	Representation	Enumeration
journal-full	Definition	A dynamic event that results when the system journals become full
	Type	nonparameter event
	Values	NOT-FULL | FULL
	Object	HSC
	Representation	Enumeration
drive-status	Definition	Status of a tape transport (tape drive)
	Type	nonparameter state
	Values	BUSY | AVAILABLE
	Object	Tape Transport
	Representation	Enumeration

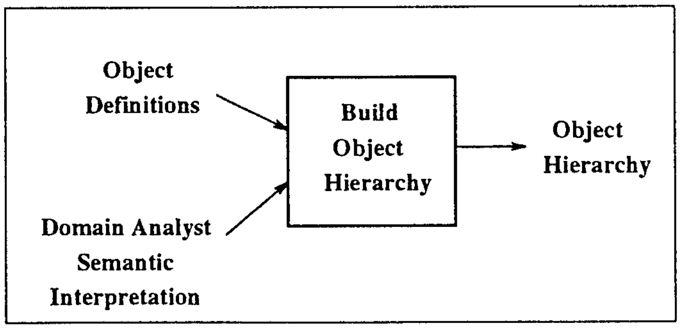


Figure 3.8: IPO Diagram : Object Hierarchy Definition

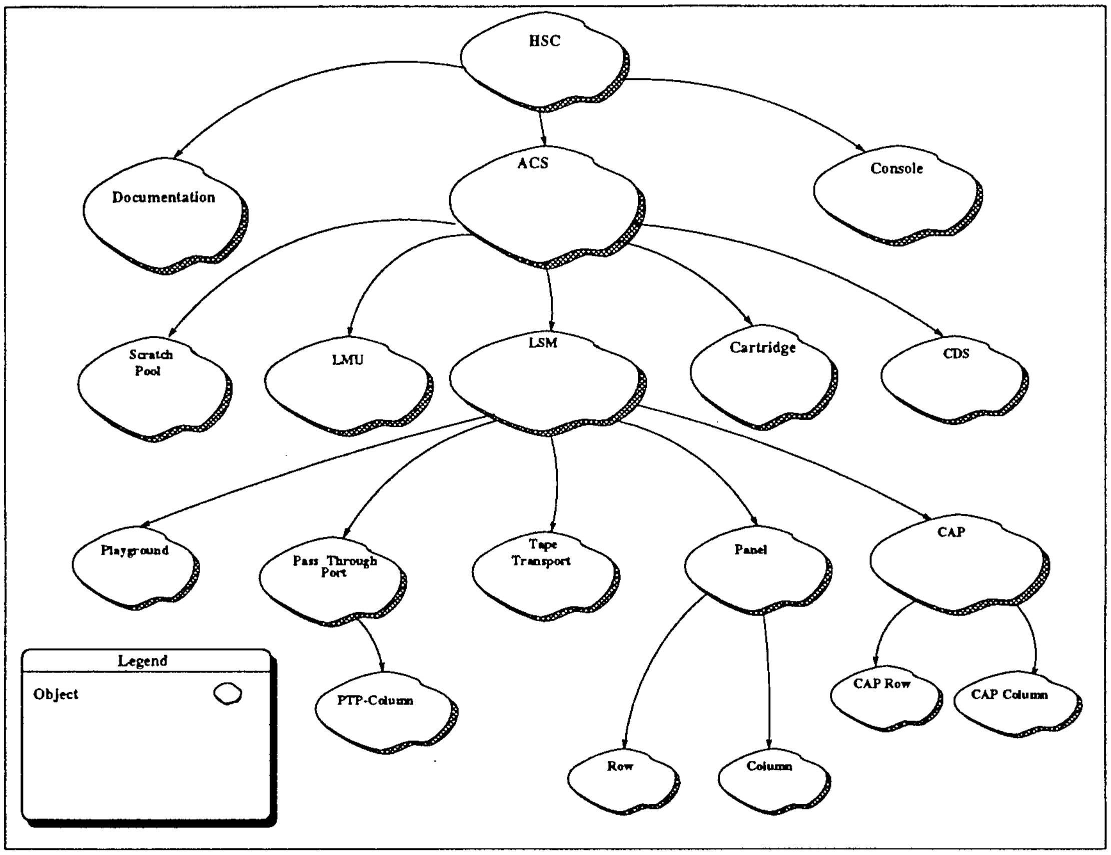


Figure 3.9: StorageTek Object Hierarchy

step, object and object element definitions, and the user's semantic interpretation of the domain. The analyst examines the parameters between "parent-child" object arcs in the object hierarchy and annotates the arcs with parameter value constraints. Figure 3.11 shows a generic object hierarchy with five objects. Relationships between objects are shown by an arrow from one object to another and parameter constraint rules are shown as labels on the arcs between the objects.

For instance, **Object1** may influence parameter values in **Object2, Object3**, or **Object4**. In the figure, the annotation :

(object1,e1) $\rightarrow$ *(object3,e3)*

shows `element1` from `Object1` has a relationship with `element3` of `Object3`. Our approach to identifying parameter constraints is:

```
FOR each (ParentObject - ChildObject) pair in the Object Hierarchy
   Let PP = set of all parent object attributes, $(PP_1, \ldots, PP_n)$
   Let CP = set of all child object attributes, $(CP_1, \ldots, CP_k)$
   FOR i=1 TO n
     FOR j=1 TO k
       IF (PP[i] constrains CP[k]) THEN
           Annotate Parent-Child Edge (PP[i] -> CP[k])
       END IF
     END FOR
   END FOR
END FOR
```

The steps examine all parent-child edges in the object hierarchy. For each parent and child, we compare each *object attribute* in the parent with all *object attributes* in the child. If a parameter constraint exists, then we annotate the edge in the object hierarchy. Evaluating whether a constraint exists requires domain analyst guidance. The analyst may also need user documentation or manuals to determine these constraints.

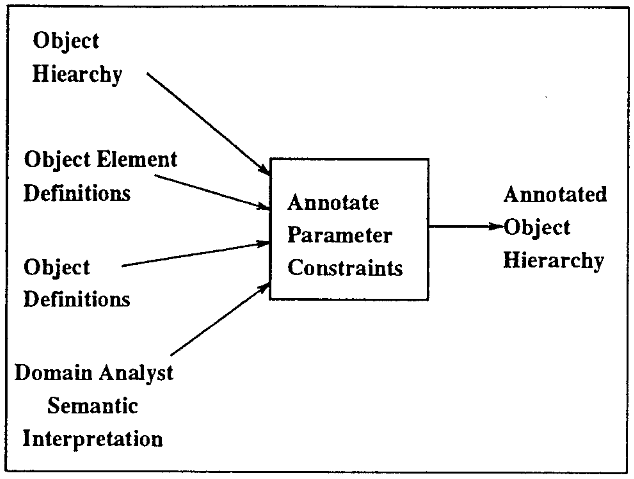

Object Hiearchy
Object Element Definitions
Object Definitions
Domain Analyst Semantic Interpretation
Annotate Parameter Constraints
Annotated Object Hierarchy

Figure 3.10: IPO Diagram : Annotated Parameter Constraints

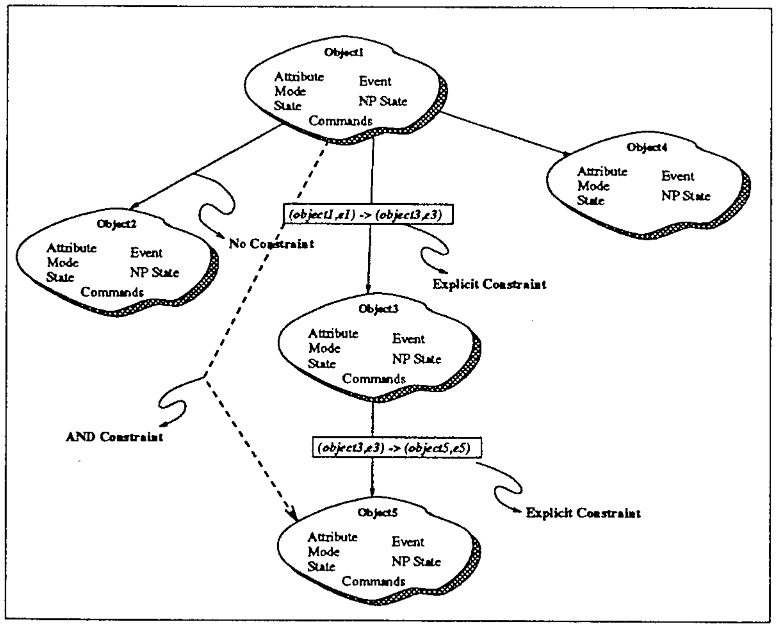

Object1
Attribute Mode State Event NP State Commands
Object2
Attribute Mode State Event NP State Commands
Object3
Attribute Mode State Event NP State Commands
Object4
Attribute Mode State Event NP State
Object5
Attribute Mode State Event NP State Commands
(object1,e1) -> (object3,e3)
(object3,e3) -> (object5,e5)
No Constraint
Explicit Constraint
AND Constraint
Explicit Constraint

Figure 3.11: Generic Object Hierarchy

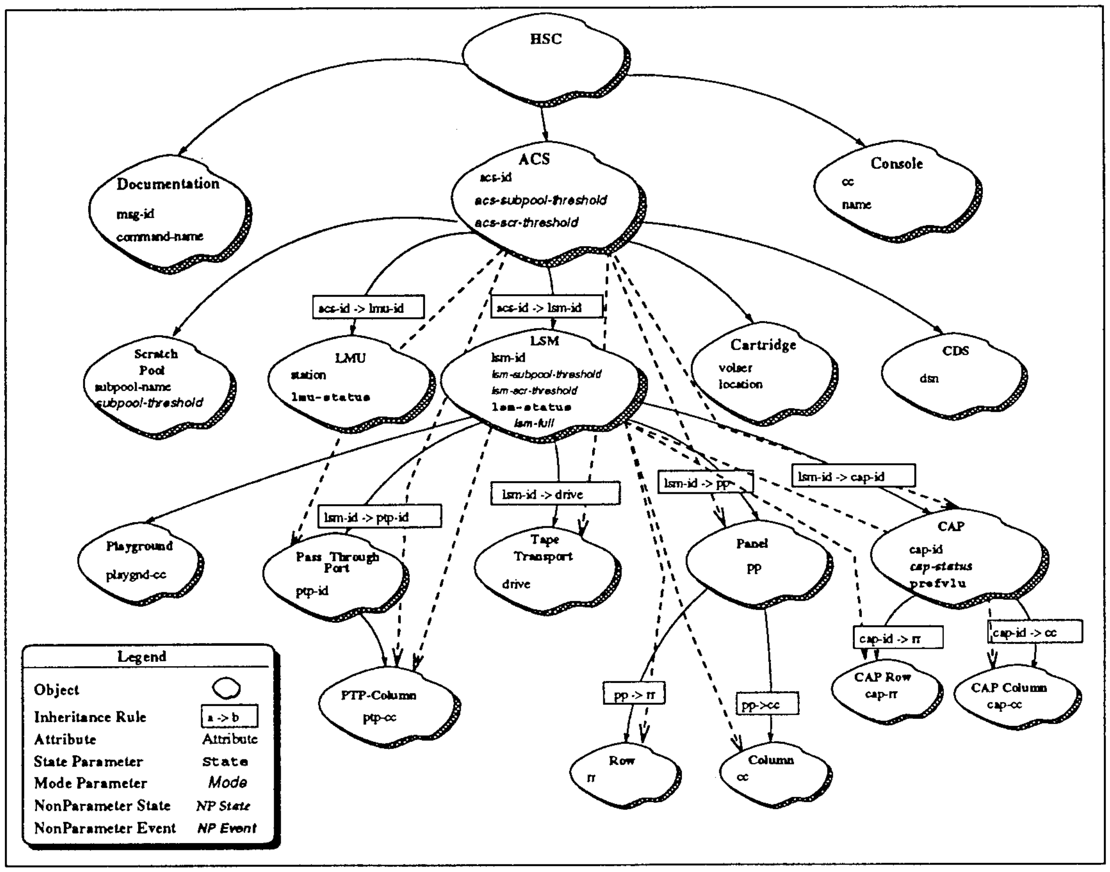


Figure 3.12: Annotated StorageTek Object Hierarchy

Figure 3.12 shows the annotated object hierarchy for the the StorageTek robot tape library. We also added detailed information to the hierarchy by showing the object elements associated with each object [1].

Types of Relationships: Table 3.9 summarizes the types of relationships between objects relevant to DBT. The first relationship, *No Constraint*, is drawn as an arrow from an object at one level of the hierarchy to an object at the next lower level with no annotation. This captures the physical relationship between objects. For instance, an ACS has a Control Data Set (CDS) database to keep track of cartridge tape locations. Despite this relationship, the *acs-id* parameter does not constrain CDS parameters.

[1]We did not list object elements for the HSC object because they would not fit nicely into the diagram. HSC has 26 object elements. The majority are *mode parameters*. HSC *modes* set various operating modes of the HSC software. See Appendix 1 for details.

Table 3.9: Types of Relationships

Type	Representation	Description
No Constraint	No annotation on the arc	Choices for the first parameter do not constraint choices for the 2nd
Explicit Constraint	$a \rightarrow b$	Parameter a constrains b
AND-Constraints	a AND $b \rightarrow c$	Explicit Constraints spanning more than one level of the hierarchy
Intraobject Constraints		Split the object into two objects

The second type, and most common, of relationship is called an *Explicit Constraint.* Here, the value of an attribute from one object constrains the values of a parameter from another object. Because this is so common, we use the notation $a \rightarrow b$ to denote object element a constrains the choices of object element b. Figure 3.12 contains many explicit constraints. For example, an LSM has panels for cartridge storage. Each LSM can be configured with different panels. Therefore, the value of the *lsm-id* parameter (an instance of an LSM) constrains the choices of the *pp* (panel number) parameter.

The third relationship, *AND-Constraints*, handles explicit constraints that span more than one level in the object hierarchy. During parameter value selection, all explicit constraints are evaluated from the root (top) of the hierarchy down to the object in question. Consider the choice of a row number (*rr*) for the HSC command language. The *AND-constraint* must consider all explicit constraints from the HSC object (root) to the Row object (object in question). An example of *AND-constraints* for a specific ACS configuration can be written as:

```
(acs-id = 00) AND (lsm-id = 000) AND (pp = 00) -> (rr = 00-05)
(acs-id = 00) AND (lsm-id = 001) AND (pp = 00) -> (rr = 00-10)
```

Figure 3.13 shows an "instantiation" of the StorageTek object hierarchy for these constraints. Note that we cannot resolve the choice for a "row" parameter by looking at the single $pp \rightarrow rr$ explicit constraint. We must include all explicit constraints from the root to the object in question.

The last object relationship is an *Intraobject Constraint.* On occasion, one may find that the range of values for one object element constraining the values of another object

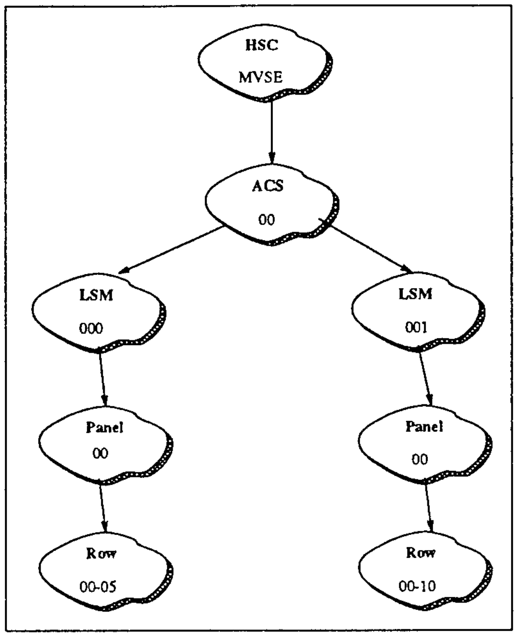


Figure 3.13: AND-Constraint Object Hierarchy Example

element *within the same object.* This *Intraobject Constraint* is removed by splitting the object and assigning each object element involved in the constraint to a different new object. The constraint between them now becomes an explicit constraint between objects. This keeps the object model uniform with regards to relationship constraints. Figure 3.14 shows the steps to split an object. Whenever an intraobject constraint is resolved by splitting an object, one must update the Object Glossary and the Object Element Glossary.

3.6 Command Definition

3.6.1 Syntax Representation

Domain analysis Step #3.1 records the syntax of each command (see Figure 3.15). DBT could use two representations for syntax, Backus Naur Form (BNF) and syntax diagrams. Consider the Command Glossary entry for the HSC `DISMOUNT` command (see Table 3.10). One field specifies the *syntax* of the command using BNF. An alternative to BNF is to use syntax diagrams. Syntax diagrams are graph representations of the syntax of the language (see Figure 3.16).

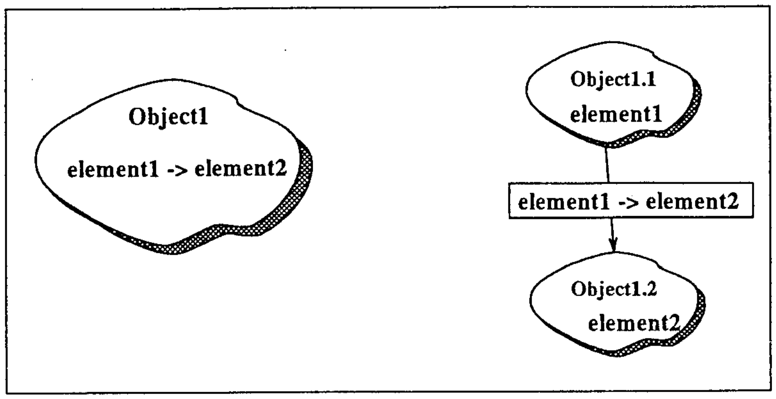


Figure 3.14: Object Splitting - What to do when constraints are local to one object

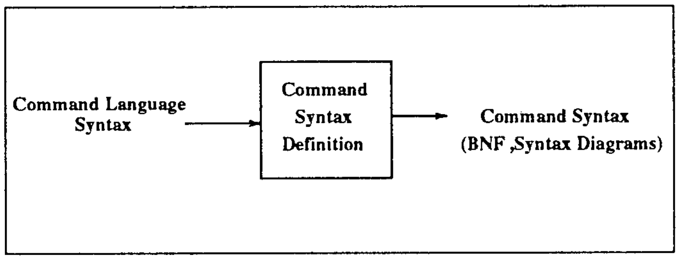

Command Language Syntax
Command Syntax Definition
Command Syntax (BNF ,Syntax Diagrams)

Figure 3.15: IPO Diagram : Command Syntax Definition

Table 3.10: Command Glossary Entry for the `DISMOUNT` command

Command Name	`DISMOUNT`
Syntax	Dismount-Cmd ::= `DISMount { , | <volser>} <drive-id> [<host-id>]`
Description	Dismount a cartridge tape from a tape drive

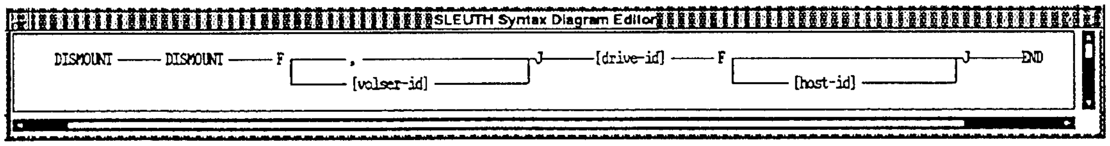


Figure 3.16: Syntax Diagram for the `DISMOUNT` Command

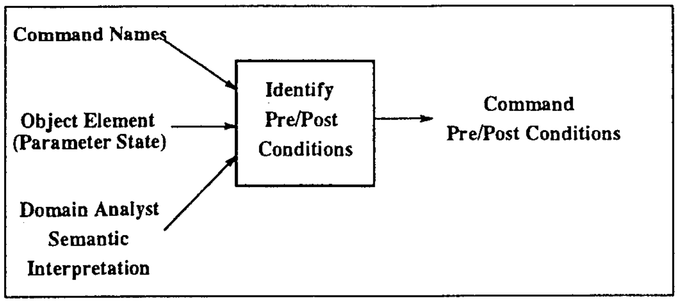

Command Names
Object Element (Parameter State)
Domain Analyst Semantic Interpretation
Identify Pre/Post Conditions
Command Pre/Post Conditions

Figure 3.17: IPO Diagram : Command Pre/Postcondition Definition

3.6.2 Identify Pre/Post Conditions

The next step in the domain analysis examines each command for preconditions and postconditions. Figure 3.17 shows that the analyst needs the list of command names, a list of object elements, and a semantic interpretation about the state required to issue each command. *Preconditions* identify the conditions that must hold before the command can execute. *Postconditions* list the conditions that are true after the command executes. Both conditions relate to the state of the objects in the system. Pre/post conditions require the analysts to examine user manuals and documentation, as well as consult their own system knowledge. The basic procedure is to associate values of *state parameter* object elements with each command as preconditions and postconditions to each command. We limit the scope to *state parameter* object elements because we can control them through the command language user interface. The recommended steps for this analysis are:

```
Let C = set of all commands, $(C_1, \ldots, C_k)$
Let PS = set of all state parameter object elements, $(PS_1, \ldots, PS_n)$
FOR i=1 to k
  FOR j=1 to n
    IF (C[i] requires PS[j] as a precondition) THEN
      enter PS[j] into C[i]'s precondition list
      enter PS[j] desired precondition value into C[i]'s precondition list
    END IF

    IF (C[i] sets PS[j] as a postcondition) THEN
      enter PS[j] into C[i]'s postcondition list
      enter PS[j] new value into C[i]'s postcondition list
    END IF
  END FOR
END FOR
```

Consider the HSC `MOUNT` command in Table 3.11. Preconditions for the command requires HSC service level = `Full`, LMU status = `Online`, and LSM status = `Online`. Suppose the preconditions were in the incorrect state. The following sequence of commands sets all preconditions such that the final `MOUNT` will be issued correctly:

```
SRVLEV Full
VARY 028 ONline
MODIFY 001 ONline
MOUNT DBT001 Readonly
```

3.6.3 Identify Intracommand Rules

The second semantic rule at the command level is called an *Intracommand Rule* (ICR). We associate these rules with a single command and use them to identify parameter constraint rules within the command. The analyst uses command syntax, and object and object element definitions to specify intracommand rules (see Figure 3.18). ICRs handle special parameter generation constraints that cannot be encoded into the object hierarchy. Parameter constraints within a command can be classified as an *exception* to the command. Analysts find them described as special notes or warnings in user documentation. Below, we formalize this analysis with a pair (2-way) parameter comparison. If we need to consider "m-way" parameter interactions, the steps can be extended.

Table 3.11: "Full" Command Glossary Entry for the HSC MOUNT Command

Command	`MOUNT`
Syntax	`Mount-cmd ::= Mount { <volser> <drive> [{,| <host-id>]} [Readonly]] | {SCRTCH|PRIVAT} <drive> [<host-id>] [SUBpool(<subpool-name>)]`
Description	Mount a cartridge tape into a tape drive
Preconditions State NP State	 Service Level = Full lmu-status(<station>) = Online lms-status(<lsm-id>) = Online
Postconditions State NP State	 drive-status(<drive-id>) = BUSY
Intracommand Rule	None.

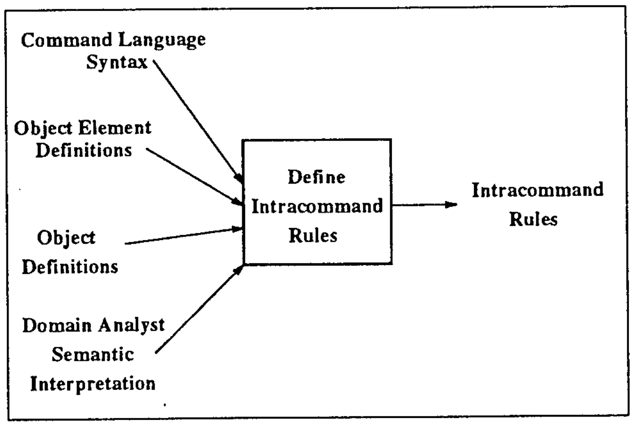

Command Language Syntax
Object Element Definitions
Object Definitions
Domain Analyst Semantic Interpretation
Define Intracommand Rules
Intracommand Rules

Figure 3.18: IPO Diagram : Intracommand Rule Definition

```
Let C = set of all commands, $(C_1, \ldots, C_k)$
FOR i=1 to k
  Let P = set of all parameters in C[i], $(P_1, \ldots, P_n)$
  FOR j=1 to n
    FOR l=j+1 to n-1
      IF (P[j] constrains P[l]) THEN
        Add the constraint to C[i]'s ICR list
      END IF
    END FOR
  END FOR
END FOR
```

To illustrate an ICR, user documentation for the StorageTek HSC `MOVE` command states, "when moving a tape within the same LSM, the source and destination panels must be different" [Tek92]. Table 3.12 shows the command glossary entry for the `MOVE` command. The domain model captures this intracommand rule as: `if (lsm-id$1=lsm-id$2) => (pp$1` $\neq$ `pp$2)`. The notation `lsm-id$1` and `lsm-id$2` denotes the first and second occurrence of the `lsm-id` parameter. If the first LSM (the source LSM) equals the second LSM (destination LSM), the source panel (pp$1) cannot equal the destination panel (pp$2).

3.7 Script Definition (Command Sequencing)

At this point in the domain analysis, we have a static model of the domain, its objects, and commands. The Scripting Definition step captures dynamic system behavior in terms of rules for sequencing commands. It also classifies commands from the problem domain. We need sequencing information because arbitrarily ordering a list of commands rarely produces semantically correct test cases. Results from an early prototype of Domain Based Testing suggests that without scripting less than 50% of the commands in the test case are meaningful [Cra93].

3.7.1 Script Class Definition

A *Script Class* groups commands according to function. Figure 3.19 shows that a domain analyst uses object definitions, command syntax, and semantic knowledge

Table 3.12: Command Glossary Entry for the HSC `MOVE` Command

Command	`Move`

Syntax	`move-cmd ::= MOVe` {FROM-LSM | VOLSER} TO-LSM
	FROM-LSM `::= Flsm(<lsm-id>) Panel(<pp>)`
	`{Row(<rr-list>) [Column(<cc>)] | Row(<rr>)`
	`[Column(<cc-list>)]`
	VOLSER `::= Volume({<volser> | <vol-range> |`
	`<vol-list>})`
	TO-LSM `::= TLsm({<lsm-id> | <lsm-list>}) [TPanel(<pp>)]`

Description	Move a tape inside the ACS
Preconditions State	 Service Level = FULL lmu-status(<station>) = Online lsm-status(<lsm-id>) = Online CDS = ENABLED
Postconditions State	 location(volsers) = changed
Intracommand Rule	• (lsm-id\$1 = lsm-id\$2) $\rightarrow$ pp\$1 $\neq$ pp\$2

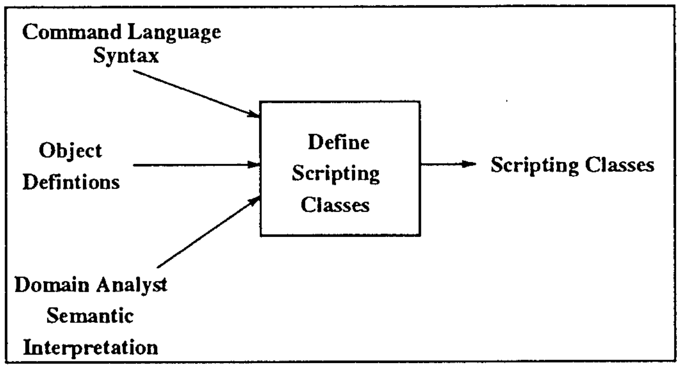


Figure 3.19: IPO Diagram : Script Class Definition

about the problem domain do define script classes. Classes provide a simple way for testers to select *what* commands to generate in a test case. The number and type of scripting classes is application dependent, and commands can be a member of more than one class. Some software products can be tested with a few classes while others may need an elaborate collection of classes.

Commands can be partitioned by function, object, and object element. Functional partitioning creates scripting classes that include commands with similar action. For example, in the StorageTek domain, the *set-up* class includes all commands that perform system set up functions; the *action* class includes commands that manipulate and

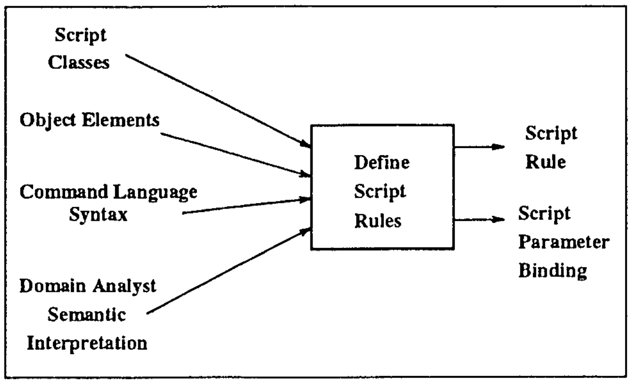


Figure 3.20: IPO Diagram : Script Rule Definition

exercise the robot tape library; and the *any* class represents the universal set that contains all commands from the command language. Partitioning commands by *object* creates classes of commands that influence a particular object. For instance, the *lsm-class* contains all commands that set up, change operating modes, or access the Library Storage Module. A script class defined as an object element partition collects commands that use a particular object element as a parameter. In the StorageTek domain, the *lsm-id-class* includes the `[Display, Modify, Move, View, Warn]` commands.

3.7.2 Script Rule Definition

Script Rules represent command sequencing semantics of the command language. In Figure 3.20, the IPO shows that the analyst needs script class names, command syntax, and object element names for this step of the domain analysis. The domain analyst uses knowledge about sequencing commands to define *scripting rules* and *script parameter bindings.*

We represent command sequences in the domain model because some commands must be issued before others. Cartridge tapes in the robot tape library must be "mounted" before they can be "dismounted." Scripts are visualized as state transition diagrams (see Figure 3.21). The script traverses various states based on the value of the current state and the choices for the next transition(s). Arcs are labeled with the names of specific commands or script classes. By restricting the commands on each

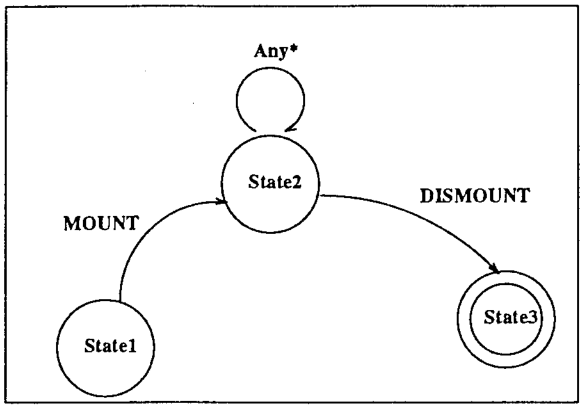


Figure 3.21: State Transition Diagram for the MOUNT-DISMOUNT Script Rule

transition, we define command sequences. The state transition diagram can be written as regular expressions. For example, the script rule in the figure could be represented as `[MOUNT Any* DISMOUNT]`.

We recommend three ways to identify script rules. The first, listed below, examines commands that influence a common object elements. Commands with common object elements could have sequencing relationships. The double arrow, $\gg$, denotes a command sequencing order. The sequence $(Cmd1 \gg Cmd2)$ shows that *Cmd-1* precedes *Cmd-2* in the script rule. For example, the `Mount-Dismount` commands influence the `drive-id` and `volser` object elements.

```
Let OE = set of all domain object elements, $(OE_1, \ldots, OE_k)$
FOR i=1 to k
  Let C = set of command that use OE[i], $(C_1, \ldots, C_n)$
  FOR i=1 to n
    FOR j=i+1 to n-1
      IF (C[i] $\gg$ C[j]) THEN
        Define C[i] $\gg$ C[j] script rule
      END IF
    END FOR
  END FOR
END FOR
```

Another way to analyze a command language for script rules is similar to the first. Instead of examining common object elements, the analyst looks for common *objects*.

Commands with common object could have sequencing relationships. The double arrow, $\gg$, denotes a command sequencing order. The sequence ($Cmd1 \gg Cmd2$) shows that *Cmd-1* precedes *Cmd-2* in the script rule. For example, the `Enter-Drain` commands influence the Cartridge Access Port (CAP) object.

```
Let O = set of all domain object, $(O_1, \ldots, O_k)$
FOR i=1 to k
  Let C = set of command that influence O[i], $(C_1, \ldots, C_n)$
  FOR i=1 to n
    FOR j=i+1 to n-1
      IF (C[i] $\gg$ C[j]) THEN
        Define C[i] $\gg$ C[j] script rule
      END IF
    END FOR
  END FOR
END FOR
```

The third approach to define scripting classes examines classes of commands divided into three functional categories: `SET-UP`, `WORK-LOAD`, and `CLEAN-UP`. These categories are common functions for most software systems. The first sets an initial system state or operating mode. The second presents a workload to the system, and the third cleans up the system and returns it to perform another task or to shut the system down. Command interaction within and between each category may require command sequencing rules. Within each category, some commands may need to be issued before others. For example, `SET-UP` command sequences are typical. Between each category, we may have command interactions, too. For instance, the effects of certain `SET-UP` commands may require certain `CLEAN-UP` actions.

At this point in the script analysis, the analyst has defined script classes and command sequencing rules. The last step is to annotate parameter binding rules for each script rule. Commands in a script rule may have constraints between their parameters. We call the constraints *script parameter binding.* Table 3.13 shows symbols used to annotate a command sequence with parameter binding rules. The first rule, $p*$, states that the value for parameter p can be selected from any valid choice as long as it fulfills parameter constraint rules. The second rule, p, restricts the value of parameter p to a

Table 3.13: Script Rule: Parameter Value Selection

Notation	Description
$p*$	Choose any valid value for p
p	Choose a previously bound value for p
p-	Choose any except a previously bound value for p

previously bound value. The third rule, p-, denotes that parameter p can be selected from any valid choice except for the currently bound value of p. To illustrate, the MOUNT - DISMOUNT sequence is annotated with script parameter selection rules.

```
MOUNT tape-id* drive-id*
Any*
DISMOUNT tape-id drive-id
```

This rule states that the *tape-id* and *drive-id* parameters can be selected from any valid choice for the MOUNT command while the DISMOUNT command must use the previously bound value for the *tape-id* and the *drive-id* parameters. Simply stated, the tape that is mounted in a drive should be dismounted from the same drive. Our analysis technique studies commands with common parameters. The steps are listed below.

```
Let C = set of all commands in script rule, $(C_1, \ldots, C_k)$
FOR i=1 to k
  FOR j=i+1 to k-1
    Let P1 = set of parameters for C[i], $(P1_1, \ldots, P1_m)$
    Let P2 = set of parameters for C[j], $(P2_1, \ldots, P2_n)$
    CP = P1 $\bigcap$ P2, $(CP_1, \ldots, CP_z)$
    FOR l=1 to z
      Annotate CP[l] with a binding symbol
    END FOR
  END FOR
END FOR
```

For each script rule, find all command names. Analyze pairs of commands by listing their common parameters, CP. For each member of CP, annotate the script rule with a parameter binding symbol. This shows how to do a pairwise analysis of the script rule. Some command languages may require "m-way" comparisons. The algorithm below can be extended for these conditions.

3.8 Summary

Domain Based Testing is a test case generation method based on domain analysis and domain modeling. In this chapter, we defined the domain analysis for command based systems. The domain analysis examines the command language at three levels of abstraction: parameter level, command level, and command sequencing level. Object and object element analysis captures parameter level domain information. At the command level, we analyze for syntax and semantic rules that apply to individual commands. In the last level, we collect semantic rules about how commands interact. Even though the steps were presented sequentially, we recommend an iterative approach to domain model definition. In the next chapter, we show how to incorporate these steps into a software reverse engineering process.

Chapter 4

DOMAIN ANALYSIS PROCESS MODEL

4.1 Introduction

The Domain Analysis Process Model (DAPM) shows how to "reverse engineer" DBT domain analysis into an existing command based system. Before starting, we must be clear on our point of view for the process. Testers could use the DBT domain analysis in two ways: a software reverse engineering effort or a new software design task. We consider the former. From a reverse engineering standpoint, we extract a domain model from an existing software application. Therefore, our domain analysis process starts with the command language definition, and completes the domain model with parameter and script level information. This is a "middle-out" analysis. In contrast, test engineers developing a new command language would follow a different trajectory through the domain analysis steps. Because the command language does not exist for a new design, they would start the domain analysis by defining "objects" and their actions (commands). Then they decide what parts of the objects need to be specified (parameters/object elements) and how to represent the command syntax.

The remainder of the chapter details the Domain Management Subsystem (DMS) (see Subsystem 0 in Figure 4.1). We expand the DMS using abstract machine diagrams (AMD) and data flow diagrams (DFD). The next section defines the symbols in each diagram. Then, we present details about three main subtasks in the DMS: *Command Definition*, *Object Definition*, and *Script Definition.*

4.2 Definitions and Symbols

We use abstract machine and data flow diagrams to describe the Domain Analysis Process Model. Figure 4.2 shows the symbols used in the abstract machine diagrams.

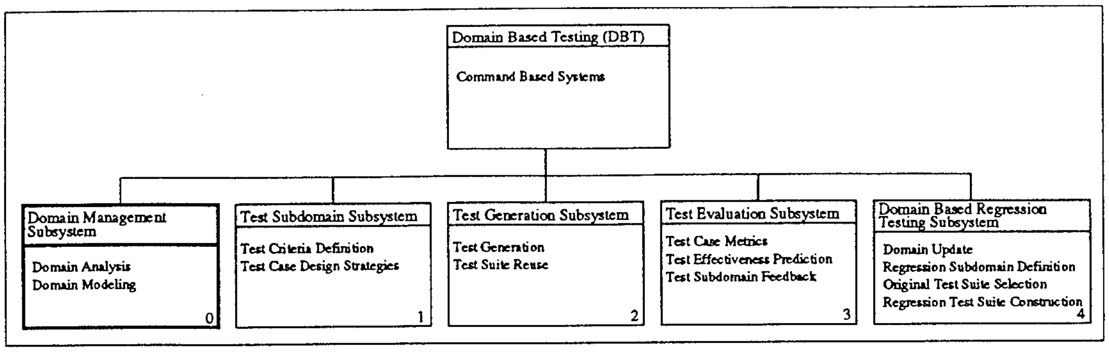

Domain Based Testing (DBT)
Command Based Systems
Domain Management Subsystem
Domain Analysis
Domain Modeling
0
Test Subdomain Subsystem
Test Criteria Definition
Test Case Design Strategies
1
Test Generation Subsystem
Test Generation
Test Suite Reuse
2
Test Evaluation Subsystem
Test Case Metrics
Test Effectiveness Prediction
Test Subdomain Feedback
3
Domain Based Regression Testing Subsystem
Domain Update
Regression Subdomain Definition
Original Test Suite Selection
Regression Test Suite Construction
4

Figure 4.1: Domain Based Testing Top Level Abstract Machine Diagram

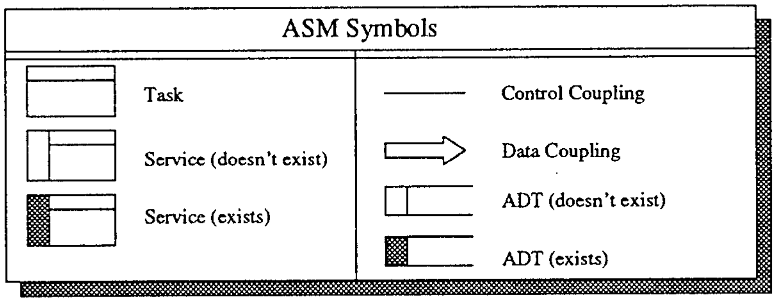


Figure 4.2: Abstract Machine Diagram Symbols

A rectangle with a title bar denotes an abstract *task*. A task combines automated and interactive services. A *service* provides a set of common utilities, functions, procedures, or libraries. Connecting *lines* from the bottom of a task denote control coupling to subtasks and services. *Outlined arrows* exiting from the side of a task denotes a "uses" relationship to abstract data types. The last symbol, an open rectangle, denotes an *abstract data type* (ADT). If a service or ADT already exists, we shade part of its symbol. Each task and service is numbered hierarchically.

Figure 4.3 lists data flow diagram symbols. A *task* in a DFD is shown as a box. The task provides an interface (automated or interactive) between the internal domain analysis tool and the "external" domain analyst. A labeled arrow defines a *flow of data* between two tasks or ADTs. Data flows use names from the data dictionary in Table 4.1 and Table 4.2 lists the regular expression symbols used in the data dictionary. An

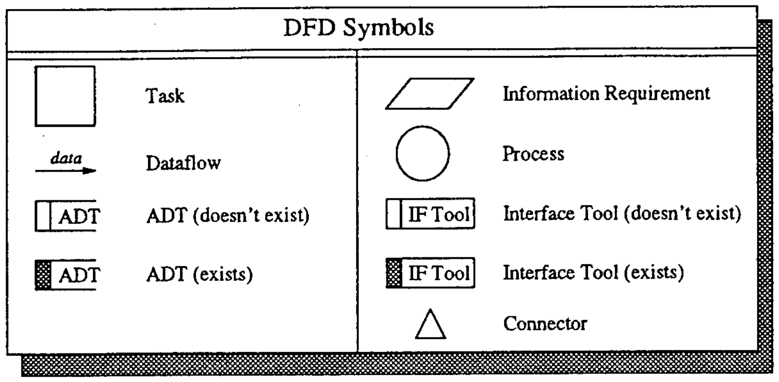


Figure 4.3: Dataflow Diagram Symbols

ADT is represented by an open rectangle, and an *information requirement* to a process is shown as a parallelogram. A circle denotes an internal, automated *process.* A closed box describes an *interface tool* between internal domain model data stores and external tasks. Because we cannot draw an entire DFD on a single page, we use a triangle to show *connections* between diagrams. If an ADT or interface tool exists, we shade part of its box. Numbers for each task and interface tool correspond to numbers in the abstract machine diagrams.

4.3 Domain Management Subsystem

Figure 4.4 expands the Domain Management Subsystem from Figure 4.1. DMS uses two subtasks: Multiple Domain Management and Domain Management. *Multiple Domain Management* (Task 0.0) provides services to create, destroy, copy, load, and save domain models. The second subtask, *Domain Management* (Task 0.1), provides utilities, editors, and automated processes to capture a domain model. The figure shows three subtasks to capture a domain model: Command Definition Task, Object Editing Task, and Script Definition Task. Each is described in the remainder of this chapter.

Figure 4.5 shows the data flow diagram (DFD) of the Domain Management Subsystem. Three processes and their associated information requirements/production describe the high level view of the domain analysis process. One way to examine a DFD is to look at the "central" process or processes. The *Command Definition Task* is the

Table 4.1: Domain Management Subsystem - Data Dictionary

Full Name	Data Name	Definition
BNF Diagram	BNFDiagram	filename
Script Class Name	Classname	string
Command Name	CmdName	string
Command Specification	CmdSpec	{CmdName | <digit/Classname>}
Intracommand Rule Record	ICRRecord	CmdName ∧ ICRRule
Intracommand Rule	ICRRule	string
Object Element Name	OEName	string
Object Element Record	OERecord	OEName ∧ OEType ∧ OEPVSet*
Object Element Parameter Value Set	OEPVSet	filename
Object Element Type	OEType	{ParmAttribute | ParmState | ParmMode | NonParmState | NonParmEvent}
Object Hierarchy Record	OHRecord	Oname ∧ Oname ∧ OHRule
Object Hierarchy Rule	OHRule	filename
Object Name	OName	string
Object Record	ORecord	OName ∧ OEName+
Parameter Bindings	ParmBindings	{OEName | OEName* | OEName-}
Postcondition	PostCondition	OEName ∧ OEType ∧ Value
Pre/Postcondition Record	PPRecord	CmdName ∧ PreCondition* ∧ PostCondition*
Precondition	PreCondition	OEName ∧ OEType ∧ Value
Script Class Record	SCRecord	ClassName ∧ CmdName+
Script Rule	ScriptRule	CmdSpec ∧ ParmBindings*
Syntax Diagram	SDiagram	filename
Syntax Record	SRecord	CmdName ∧ SDiagram ∧ BNFDiagram
Script Rule Record	SRRecord	CmdName ∧ ScriptRule+
Pre/Postcondition Value	Value	string

Table 4.2: Data Dictionary Symbols

Data Dictionary Symbols	
Symbol	Definition
a+	One or more copies of 'a'
a*	Zero or more copies of 'a'
a ∧ b	'a' and 'b' concatenated
{a | b}	Choose 'a' or 'b'

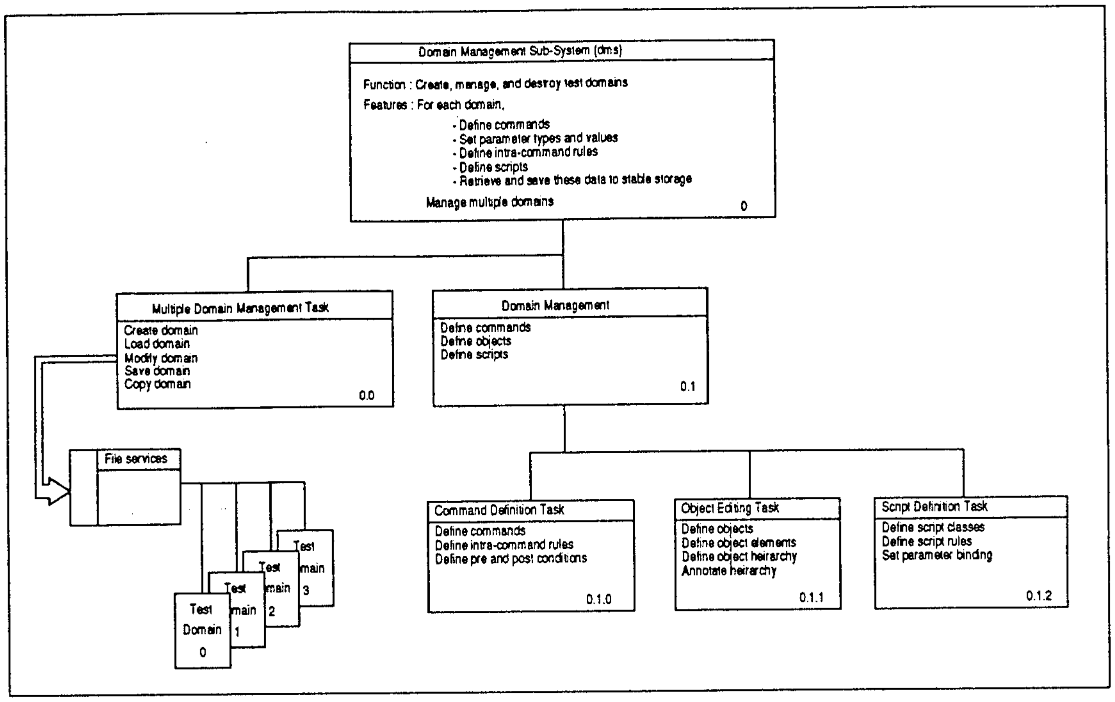


Figure 4.4: AMD Level 0.1 - Domain Management Subsystem

central process for the DMS. This should be expected because we define the domain analysis process as a reverse engineering effort, and we start with an existing command language. From the command language syntax, analysts define object elements, objects, and scripts.

4.4 Command Definition Task

Testers start the DBT domain analysis process at the *Command Definition Task* (Task 0.1.0). Figure 4.6 shows its abstract machine. The command definition task contains three subtasks: Command Syntax Definition, Intracommand Rule Definition, and Pre/Post Condition Definition. Command Syntax Definition (Task 0.1.0.0) uses an interface tool to capture command language syntax. We currently use a syntax diagram editor but this tool could be replaced with a BNF or grammar editor. Syntax information is stored in a *Command Syntax Table* (CST) where each entry is "keyed" by command name. The second subtask, Intracommand Rule (ICR) Definition (Task 0.1.0.1), captures parameter constraint rules associated with a single command. ICR

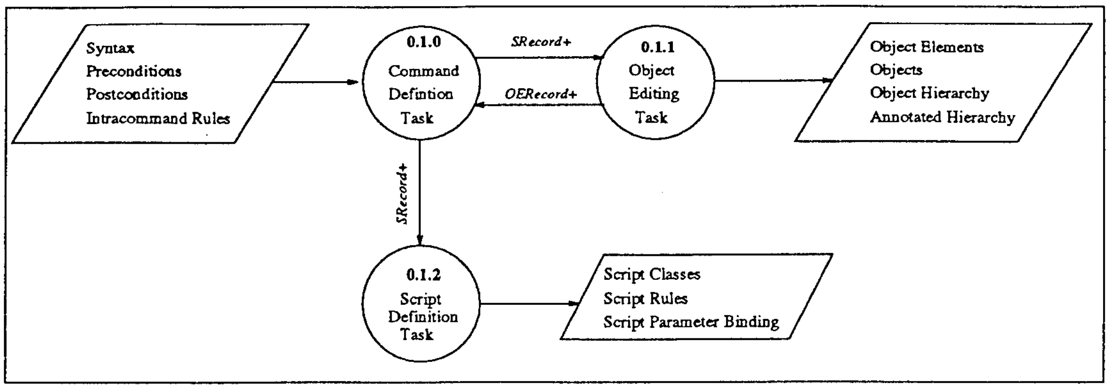


Figure 4.5: DFD Level 0.1 - Domain Management Subsystem

information is stored in an *Intracommand Rule Table* (IRT) where each entry is "keyed" by command name. The third Command Definition subtask is Pre/Post Condition Definition (Task 0.1.0.2). This task captures preconditions and postconditions for each command using a pre/post condition editor and a automatic object element extraction service. The Parameter State Extraction Service (Task 0.1.0.2.0) provides a list of candidate "state parameters" to annotate pre/post conditions for each command. The pre/post condition editor provides an interactive interface to capture the information from a test engineer. All pre/post condition information is stored in the *Pre/Post Conditions Table* (PCT) keyed by command name.

Figure 4.7 shows the data flow diagram view of the Command Definition Task. It shows the tasks and services from the AMD and the data flow information between each component. The DFD also shows connections to other data flow diagrams indicating coupling to other domain analysis steps and domain model components. The *Syntax Diagram Editor* is the central task in the Command Definition Subsystem. Because they assume the system to be tested, and thus the syntax of the commands exists, testers must "boot-strap" the process by entering command language syntax. Once boot-strapped, the domain analysis process embellishes the command language with pre/post conditions, intracommand rules, object definitions, or script definitions.

We designed the domain analysis process to be an iterative, incremental domain model capture. For instance, testers enter the syntax of a few commands, extract object

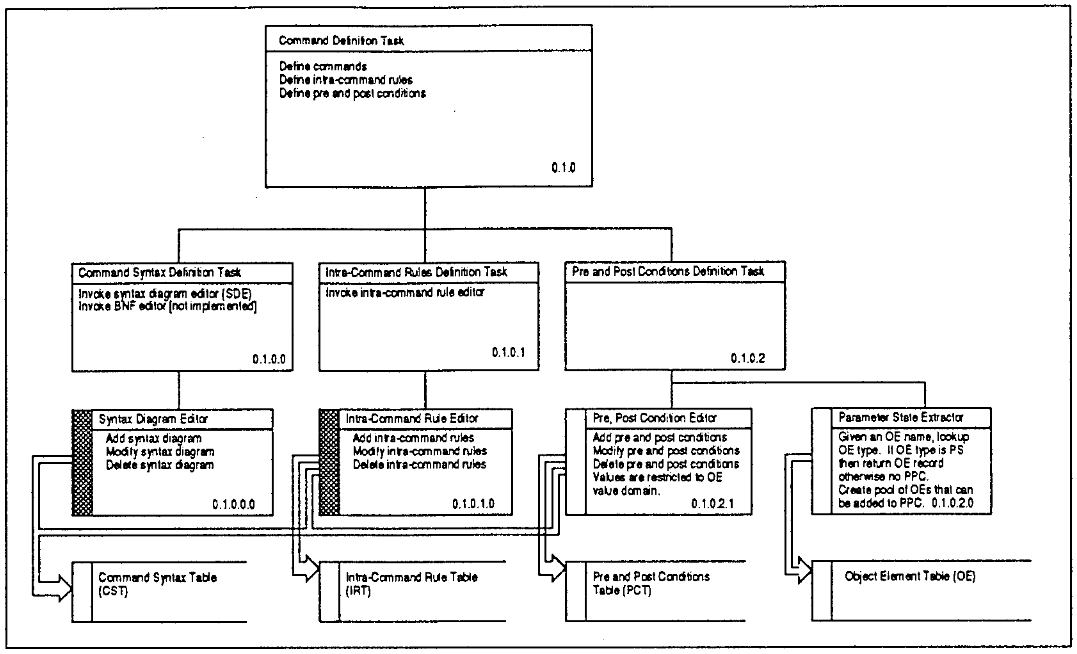

Command Definition Task
Define commands
Define intra-command rules
Define pre and post conditions
0.1.0
Command Syntax Definition Task
Invoke syntax diagram editor (SDE)
Invoke BNF editor [not implemented]
0.1.0.0
Intra-Command Rules Definition Task
Invoke intra-command rule editor
0.1.0.1
Pre and Post Conditions Definition Task
0.1.0.2
Syntax Diagram Editor
Add syntax diagram
Modify syntax diagram
Delete syntax diagram
0.1.0.0.0
Intra-Command Rule Editor
Add intra-command rules
Modify intra-command rules
Delete intra-command rules
0.1.0.1.0
Pre, Post Condition Editor
Add pre and post conditions
Modify pre and post conditions
Delete pre and post conditions
Values are restricted to OE value domain.
0.1.0.2.1
Parameter State Extractor
Given an OE name, lookup OE type. If OE type is PS then return OE record otherwise no PPC.
Create pool of OEs that can be added to PPC. 0.1.0.2.0
Command Syntax Table (CST)
Intra-Command Rule Table (IRT)
Pre and Post Conditions Table (PCT)
Object Element Table (OE)

Figure 4.6: AMD Level 0.1.0 - Command Definition

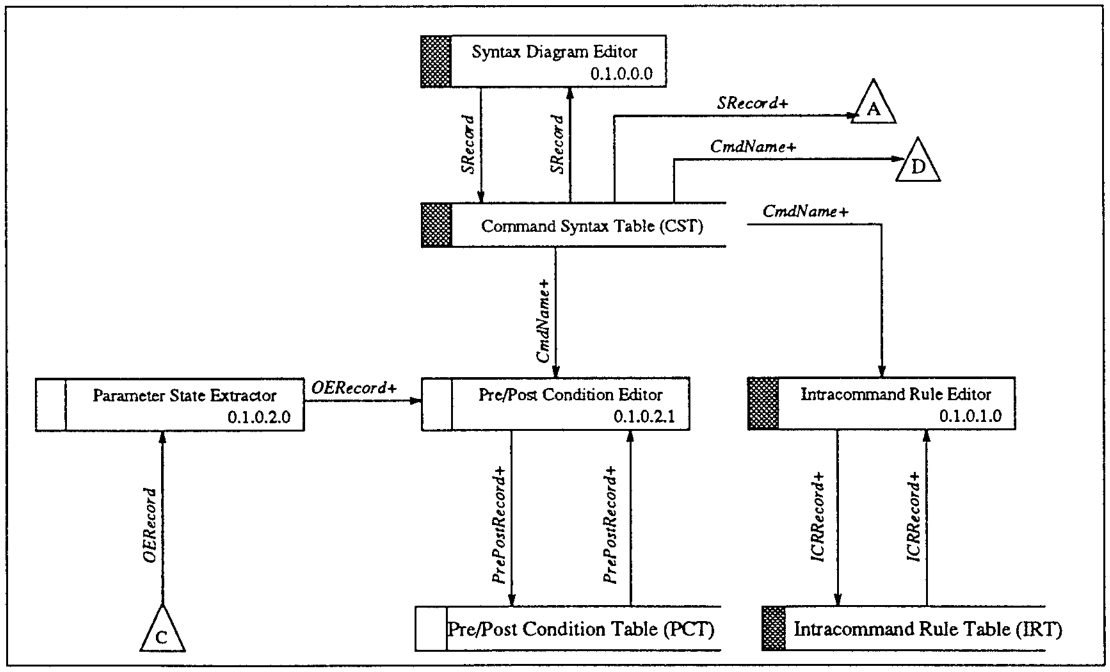


Figure 4.7: DFD Level 0.1.0 - Command Definition

elements, and define objects. Later, they may add new command syntax or modify the syntax of existing commands. To keep domain model information consistent, the incremental domain model capture forces testers to use particular editors or restricts DMS services at certain times. Consider a new command language with no domain model. The domain analysis process requires the tester to enter command language syntax for at least one command. Given the syntax of at least one command, the process allows the tester to use other services (such as *Modify Command Syntax*, *Add Intracommand Rule*, or *Define Object Elements*).

4.5 Object Editing Task

The *Object Editing Task* (Task 0.1.1) is the most complex subtask in the Domain Management Subsystem. Figure 4.8 shows the first "leveling" of object editing into three subtasks: Object Element Definition (0.1.1.0), Object Definition Task (0.1.1.1), and Object Hierarchy Definition (0.1.1.2). Figure 4.9 shows the detailed AMD view of the object element definition and object definition subtasks. Object Element Definition uses a *Syntax Extraction Service* (Service 0.1.1.0.0). This service automatically identifies parameters from command language syntax, and it enters each name into the *Object Element Table* (OET). We call command language parameters object elements, and we categorize them by object element type, and parameter value set to assist in domain model construction, test subdomain definition, and test case generation. A second service called the *Object Element Editor* (Service 0.1.1.0.1) provides a user interface to add, delete, and modify object element information. The editor uses services to annotate each object element name with an object element type and a parameter value set.

Object Element Definition uses services and automatic processes to control incremental domain model capture. For instance, each time the Object Element Definition runs, the Syntax Extraction Process scans the command syntax for command language parameters. These parameters are automatically entered into the Object Element Table (OET). This guarantees up-to-date object element information.

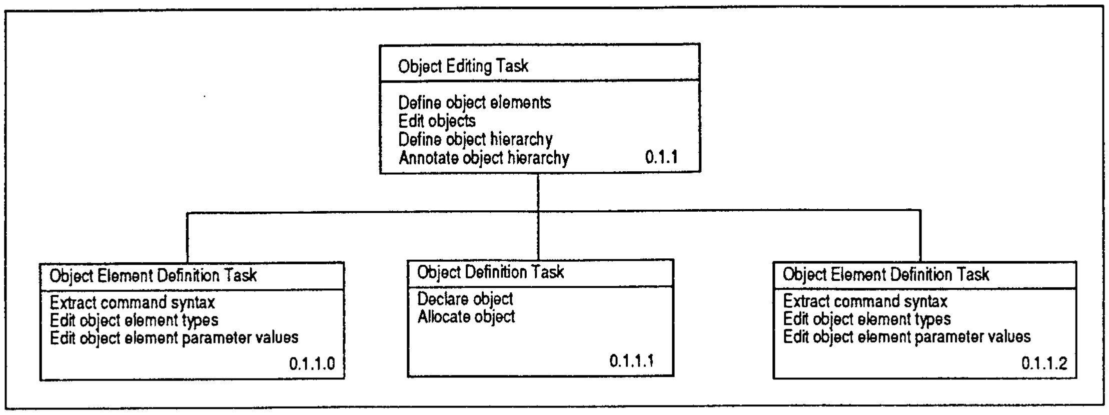

Object Editing Task
Define object elements
Edit objects
Define object hierarchy
Annotate object hierarchy
0.1.1
Object Element Definition Task
Extract command syntax
Edit object element types
Edit object element parameter values
0.1.1.0
Object Definition Task
Declare object
Allocate object
0.1.1.1
Object Element Definition Task
Extract command syntax
Edit object element types
Edit object element parameter values
0.1.1.2

Figure 4.8: AMD Level 0.1.1

Figure 4.9 also shows the *Object Definition Task* (Task 0.1.1.1). Testers use the object definition utilities to define objects of the system under test and to populate the objects with object elements. The object definition task uses two services: Object Declaration Editor (Service 0.1.1.1.0.0) and the Object Allocation Editor (Service 0.1.1.1.0.1). The Object Declaration Editor provides a user interface to define, add, delete, and modify objects. We store all objects in an *Object Table* (OT) keyed by object name. The Object Allocation Editor reads the Object Element Table, extracts object element names, and provides utilities to associate object elements with objects.

Figure 4.10 shows the data flow digram for the object element and the object definition tasks. While combined, the DFD has a natural "split" between the two tasks. Hence, we see two central processes in the DFDs: Object Element Editor (Service 0.1.1.0.1) and the Object Editor (Service 0.1.1.1.0). Connector *A* shows the data flow connection to the *Command Syntax Table.* Connectors *B* and *C* show that object element and object information is needed in other parts of the domain management architecture.

The third part of the *Object Definition Task* captures object relationships and parameter constraint rules. The *Object Hierarchy Definition Task* (Task 0.1.1.2) uses two services: Hierarchy Extractor (Service 0.1.1.2.0) and the Hierarchy Editor (Service 0.1.1.2.1) (see Figure 4.11). The first service extracts object names from the *Object*

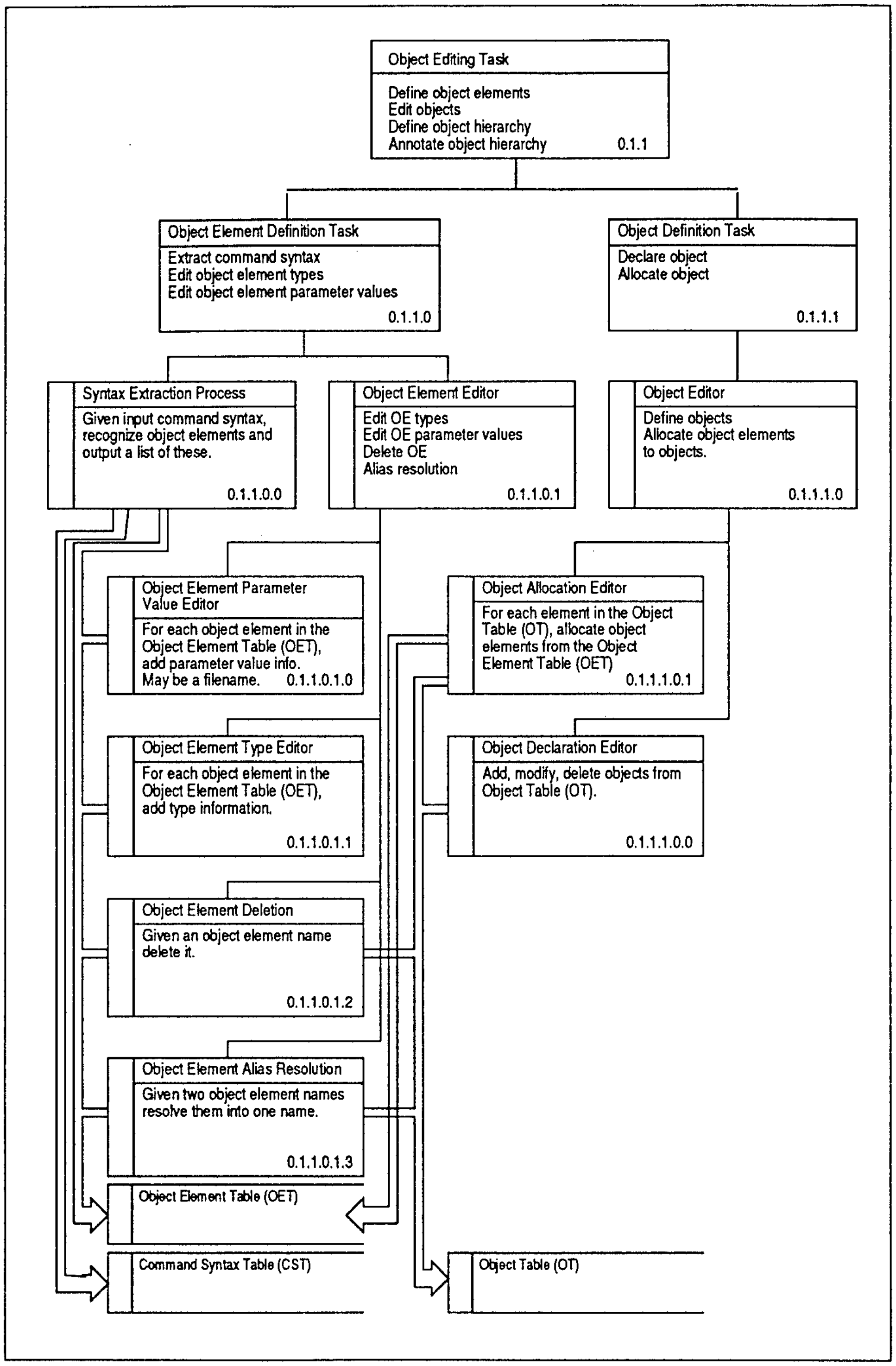

Object Editing Task
Define object elements
Edit objects
Define object hierarchy
Annotate object hierarchy 0.1.1
Object Element Definition Task
Extract command syntax
Edit object element types
Edit object element parameter values
0.1.1.0
Object Definition Task
Declare object
Allocate object
0.1.1.1
Syntax Extraction Process
Given input command syntax, recognize object elements and output a list of these.
0.1.1.0.0
Object Element Editor
Edit OE types
Edit OE parameter values
Delete OE
Alias resolution
0.1.1.0.1
Object Editor
Define objects
Allocate object elements to objects.
0.1.1.1.0
Object Element Parameter Value Editor
For each object element in the Object Element Table (OET), add parameter value info. May be a filename. 0.1.1.0.1.0
Object Allocation Editor
For each element in the Object Table (OT), allocate object elements from the Object Element Table (OET) 0.1.1.1.0.1
Object Element Type Editor
For each object element in the Object Element Table (OET), add type information.
0.1.1.0.1.1
Object Declaration Editor
Add, modify, delete objects from Object Table (OT).
0.1.1.1.0.0
Object Element Deletion
Given an object element name delete it.
0.1.1.0.1.2
Object Element Alias Resolution
Given two object element names resolve them into one name.
0.1.1.0.1.3
Object Element Table (OET)
Command Syntax Table (CST)
Object Table (OT)

Figure 4.9: AMD Level 0.1.1 - Object Definition (Part 1)

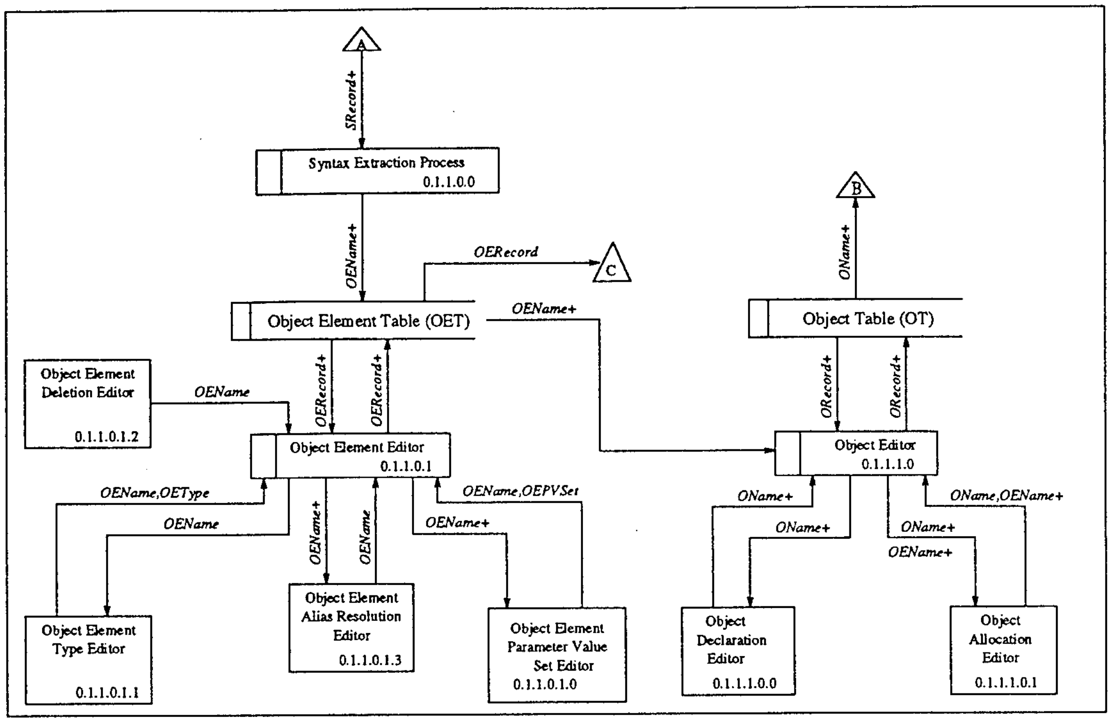

A
SRecord+
Syntax Extraction Process 0.1.1.0.0
OEName+
OERecord
C
B
OName+
Object Element Table (OET)
OEName+
Object Table (OT)
Object Element Deletion Editor 0.1.1.0.1.2
OEName
OERecord+
OERecord+
ORecord+
ORecord+
Object Element Editor 0.1.1.0.1
Object Editor 0.1.1.1.0
OEName,OEType
OEName,OEPVSet
OEName
OEName+
OEName
OEName+
ONameр+
ONamе+
ONamе,OEName+
ONamе+
OEName+
Object Element Type Editor 0.1.1.0.1.1
Object Element Alias Resolution Editor 0.1.1.0.1.3
Object Element Parameter Value Set Editor 0.1.1.0.1.0
Object Declaration Editor 0.1.1.1.0.0
Object Allocation Editor 0.1.1.1.0.1

Figure 4.10: DFD Level 0.1.1 - Object Definition (Part 1)

Table and reads the current object hierarchy from the *Object Hierarchy Table* (OHT). This service provides a pool of objects to place in the current hierarchy. The pool of objects may be *full* when the Hierarchy Editor runs on a domain model for the first time. The object pool may be *empty* if all objects are currently placed in the hierarchy. The Object Hierarchy Editor uses the current hierarchy and the pool of objects in an interactive editor to manipulate the object hierarchy and to annotate the hierarchy with parameter constraint rules. The Object Hierarchy Editor employs a variety of tools, services, and consistency checking processes. All object hierarchy information is stored in the *Object Hierarchy Table* keyed by "parent-child" pairs.

Figure 4.12 shows the data flow diagram for the object hierarchy definition. The central task is the *Object Hierarchy Editor.* The hierarchy extractor and consistency check processes are automated, internal mechanisms to check and maintain an accurate object hierarchy. The *Object Relationship Editor* (Service 0.1.1.2.1.0) and the *Parameter Constraint Editor* (Service 0.1.1.2.1.2) provide user interface tools and services to capture object hierarchy and parameter constraint information from the domain analyst.

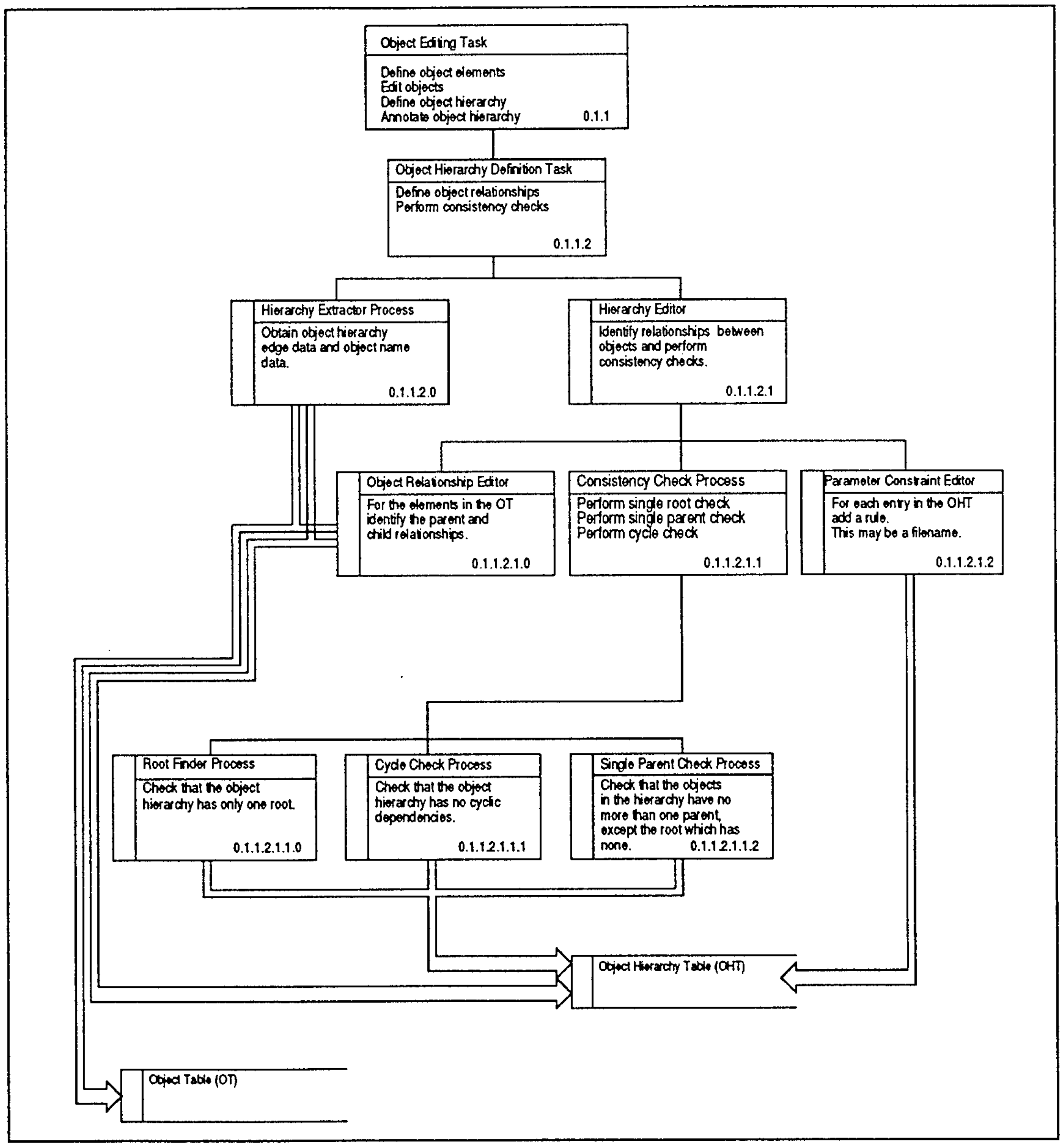

Object Editing Task
Define object elements
Edit objects
Define object hierarchy
Annotate object hierarchy
0.1.1
Object Hierarchy Definition Task
Define object relationships
Perform consistency checks
0.1.1.2
Hierarchy Extractor Process
Obtain object hierarchy edge data and object name data.
0.1.1.2.0
Hierarchy Editor
Identify relationships between objects and perform consistency checks.
0.1.1.2.1
Object Relationship Editor
For the elements in the OT identify the parent and child relationships.
0.1.1.2.1.0
Consistency Check Process
Perform single root check
Perform single parent check
Perform cycle check
0.1.1.2.1.1
Parameter Constraint Editor
For each entry in the OHT add a rule.
This may be a filename.
0.1.1.2.1.2
Root Finder Process
Check that the object hierarchy has only one root.
0.1.1.2.1.1.0
Cycle Check Process
Check that the object hierarchy has no cyclic dependencies.
0.1.1.2.1.1.1
Single Parent Check Process
Check that the objects in the hierarchy have no more than one parent, except the root which has none.
0.1.1.2.1.1.2
Object Hierarchy Table (OHT)
Object Table (OT)

Figure 4.11: AMD Level 0.1.1 - Object Definition (Part 2)

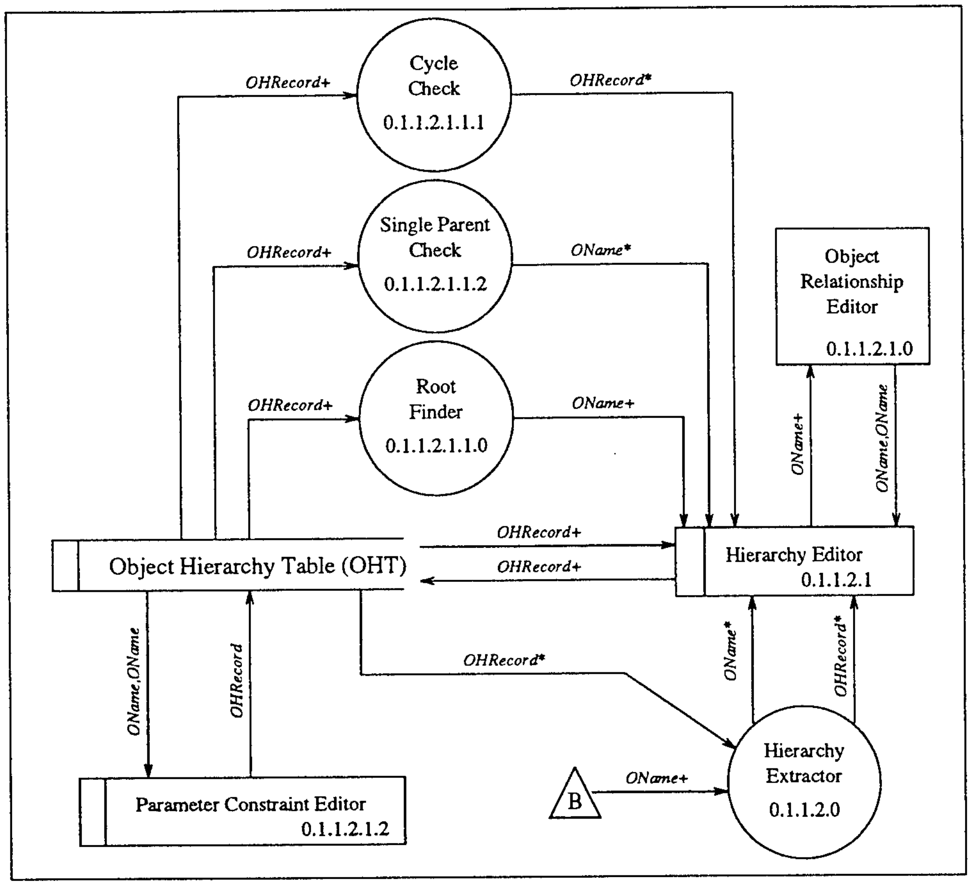


Figure 4.12: DFD Level 0.1.1 - Object Definition (Part 2)

4.6 Script Definition Task

Figure 4.13 shows the abstract machine for the *Script Definition Task* (Task 0.1.2). Script Definition uses two services: Script Class Editor and Script Rules Editor. The *Script Class Editor* (Service 0.1.2.0) provides services to define, modify, and delete script class names from the domain model. Domain analysts also assign command names to the script classes. Script classes are stored in a *Script Class Table* (SCT) keyed by script class name. The *Script Allocation Editor* reads the *Command Syntax Table* (CST) creating a pool of command names to assign to script classes. The Script Allocation Editor automatically defines the `ANY` class as the universal set of all commands in the command language.

The second service, *Script Rules Editor* (Service 0.1.2.1), uses two sub-services: *Script Rule Editor* and *Parameter Binding Editor.* The Script Rule Editor (Service 0.1.2.10) provides a user interface to capture command sequencing information from the domain analyst. All scripting rules are stored in the *Script Rule Table* (SRT) keyed by command name. The Parameter Binding Editor (Service 0.1.2.1.1) annotates script rules with parameter binding information. The editor employs the services of the Parameter List Generator (Service 0.1.2.1.1.0). The list generator reads the Command Syntax Table and prepares a list of object elements in common to the commands in the script rule. Common object elements are candidates for parameter binding rules.

Figure 4.14 shows the data flow diagram for the Script Definition Task. The Script Declaration Editor and the Script Rule Editor are the central DFD processes. The Script Declaration Editor serves as an interface between the SCT and the Script Allocation Editor. The Script Rule Editor combines information about command sequencing, script classes, and parameter binding. All script rule information is stored in the Script Rules Table. Each time the Script Definition Task runs, editors and services retrieve the latest information from the ADTs. The Script Class Editor retrieves current script class definitions from the Script Class Table ADT. The Script Rules Editor consults the Script Rule Table ADT for the current list of script rules. When new script classes

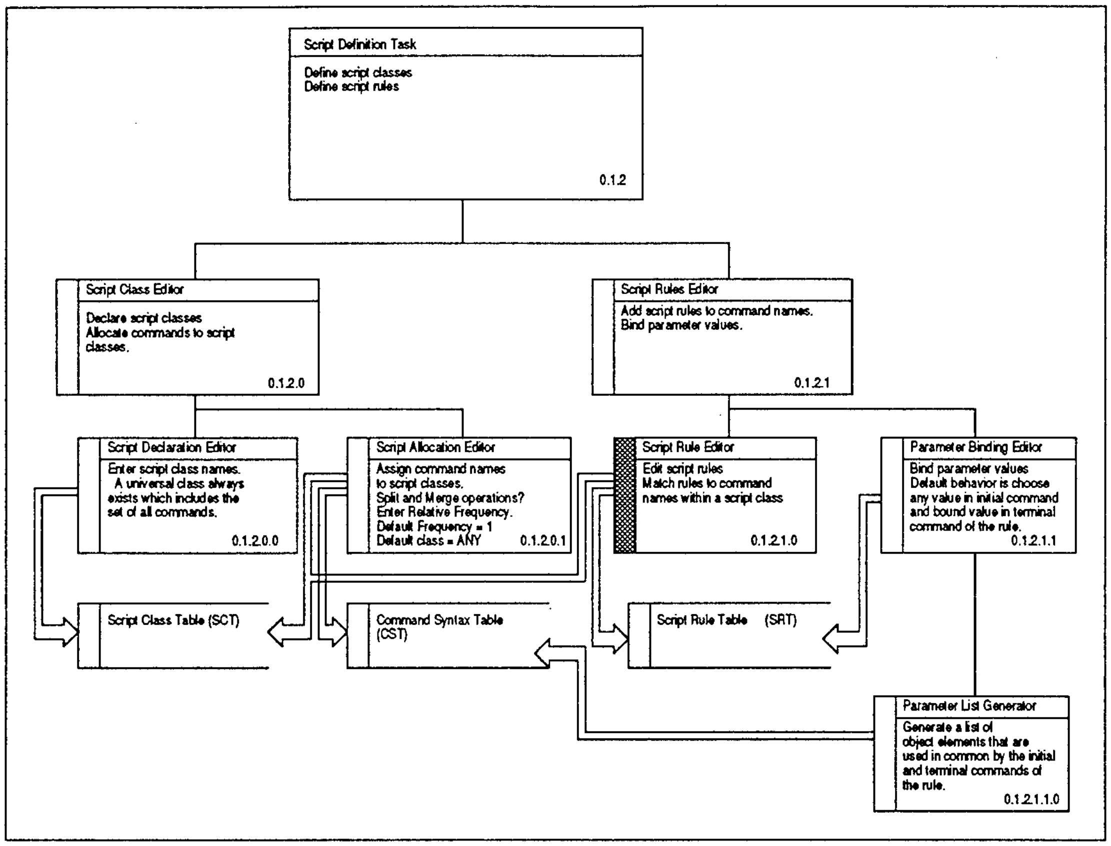

Script Definition Task
Define script classes
Define script rules
0.1.2
Script Class Editor
Declare script classes
Allocate commands to script classes.
0.1.2.0
Script Rules Editor
Add script rules to command names.
Bind parameter values.
0.1.2.1
Script Declaration Editor
Enter script class names.
A universal class always exists which includes the set of all commands.
0.1.2.0.0
Script Allocation Editor
Assign command names to script classes.
Split and Merge operations?
Enter Relative Frequency.
Default Frequency = 1
Default class = ANY
0.1.2.0.1
Script Rule Editor
Edit script rules
Match rules to command names within a script class
0.1.2.1.0
Parameter Binding Editor
Bind parameter values
Default behavior is choose any value in initial command and bound value in terminal command of the rule.
0.1.2.1.1
Script Class Table (SCT)
Command Syntax Table (CST)
Script Rule Table (SRT)
Parameter List Generator
Generate a list of object elements that are used in common by the initial and terminal commands of the rule.
0.1.2.1.1.0

Figure 4.13: AMD Level 0.1.2 - Script Definition

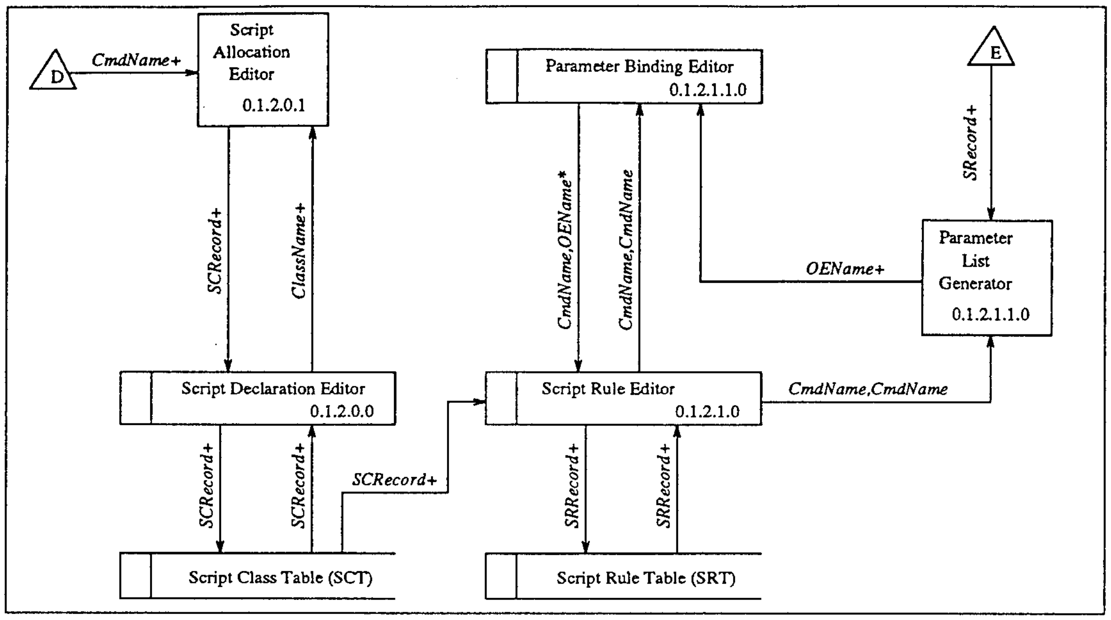


Figure 4.14: DFD Level 0.1.2 - Script Definition

or script rules are defined, the Script Definition Task relies on the Script Allocation Editor, Parameter Binding Editor, and the Parameter List Generator Services to consult the latest information in the Command Syntax Table ADT. This design supports incremental domain model capture. Suppose a test engineer enters syntax of new commands into the Command Syntax Table. The Script Allocation Editor reads the CST, automatically updates the ANY class, and provides utilities to assign command names to script classes. The result is a Script Definition Task that always uses the most current command language information.

4.7 Conclusions

The Domain Analysis Process Model (DAPM) provides an architectural design for a domain analysis tool for DBT. This chapter describes a reverse engineering approach for DBT. Command, Script, and Object tasks were specified using abstract machine diagrams and data flow diagrams. The result is a process model for an "incremental domain model capture" of an existing software system with a command language interface.

Chapter 5

TEST GENERATION PROCESS MODEL

5.1 Introduction

The Test Generation Process Model couples the domain model with a test generation process. Our goal is to automate Domain Based Testing and to provide a useful technology transfer of these ideas to industry. The test generation process shows how testers use an automated DBT test generator, identifies steps testers need to control, and isolates fully automated test generation functions. This chapter describes two subsystems in the DBT Architecture (see Subsystems 1 and 2 Figure 5.1). The Test Subdomain Subsystem shows how to incorporate test criteria into the domain model, and the Test Generation Subsystem uses the test subdomain for test data generation.

The DBT Test Generation Process is modeled from a tester's work process. Working with an industrial test team, we investigated (1) How they tested their products, (2) The steps in their testing process, (3) What automated testing tools they needed, and (4) What features would they require in an automated test generator. The results from this inquiry shows that the testers used a four step process:

```
1. Choose a system component to test.
2. Define a test criteria for the test.
3. Determine a set of commands to exercise the component.
4. Generate a sequence of commands using parameters from a
   specific system configuration.
```

From this investigation, we designed a DBT Test Generation Process based on the domain model representation and the four step testing process (see Figure 5.2) The DBT test generation process has four components *domain model*, *test subdomain definition*, *test criteria*, and *test generation*. First, a *domain model* is defined for the system under

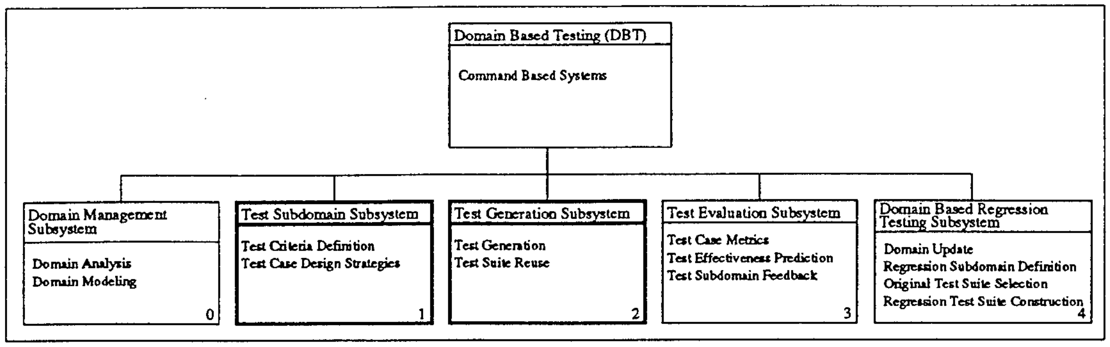

Domain Based Testing (DBT)
Command Based Systems
Domain Management Subsystem
Domain Analysis
Domain Modeling
0
Test Subdomain Subsystem
Test Criteria Definition
Test Case Design Strategies
1
Test Generation Subsystem
Test Generation
Test Suite Reuse
2
Test Evaluation Subsystem
Test Case Metrics
Test Effectiveness Prediction
Test Subdomain Feedback
3
Domain Based Regression Testing Subsystem
Domain Update
Regression Subdomain Definition
Original Test Suite Selection
Regression Test Suite Construction
4

Figure 5.1: Domain Based Testing Top Level Abstract Machine Diagram

test. Analysts use the process outlined in the previous chapters to create the domain model. Next, test engineers modify the domain model to create *test subdomains.* Testers configure a test subdomain for specific test scenarios or test strategies. *Test Generation* uses information from the test subdomain and test criteria to create test suites. A *test suite* for DBT contains *test cases*, *test templates*, and *test scripts.* A test case is a list of fully parameterized commands from the syntax of the problem domain. A test template is a list of commands with place holders for parameters, and test scripts are lists of command names.

5.2 Test Generation Process Diagram

The IPO diagram in Figure 5.3 shows the input and output for each step in the test generation process. The IPO adds detail to Figure 3.3 from Chapter 3. Table 5.1 defines each set in the figure. The domain model D_0^v captures the syntax and semantics of the system under test. The zero subscript identifies the domain model as the starting point from which all tests are generated. The superscript v identifies the version of the system under test. For instance, $D_0^{R1.2}$ denotes the StorageTek Release 1.2 Domain Model. A domain is a persistent view of the system because it represents the default conditions to generate test suites. A domain model is needed for each new domain and every time a domain changes significantly. All testers share the domain model to provide a consistent view of the system under test. Sometimes test objectives call for

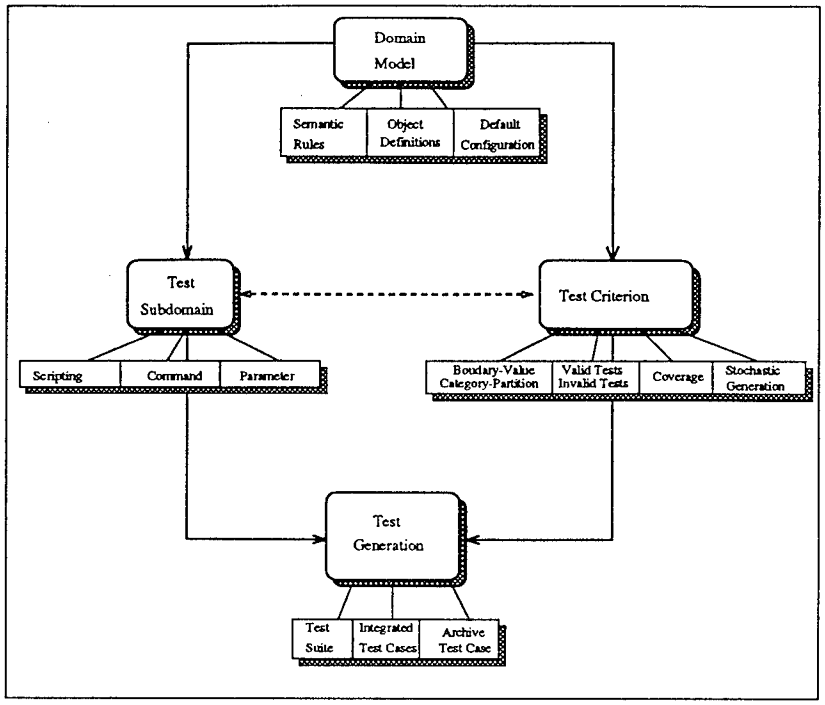


Figure 5.2: Test Generation Process Model

Table 5.1: DBT Definitions

Set	Definition
D_0^v	Domain Model for version v
TSD_j^v	Test Subdomain j for version v
T_{j-k}^v	Test Suite $k (k = 1, 2, 3, \ldots)$ for version v and subdomain j

test cases generated directly from D_0^v. Such tests are "valid" sequences of commands that follow all syntax and semantic rules defined in D_0^v.

Most of the time, testers modify the domain model to test a system configuration or to test a particular feature. Any change to the domain model defines a *test subdomain*, TSD_j^v. Modifications to a domain model include: restricting the set of commands that can be generated, turning On/Off semantic rules, or changing parameter value sets and parameter constraint rules. Each test subdomain is specified by a subscript and a superscript. The subscript j identifies the specific subdomain created, and the superscript identifies the version of the system under test. For example, $TSD_{CAP}^{R1.2}$ is the Cartridge Access Port (CAP) Test Subdomain for StorageTek Release 1.2. All

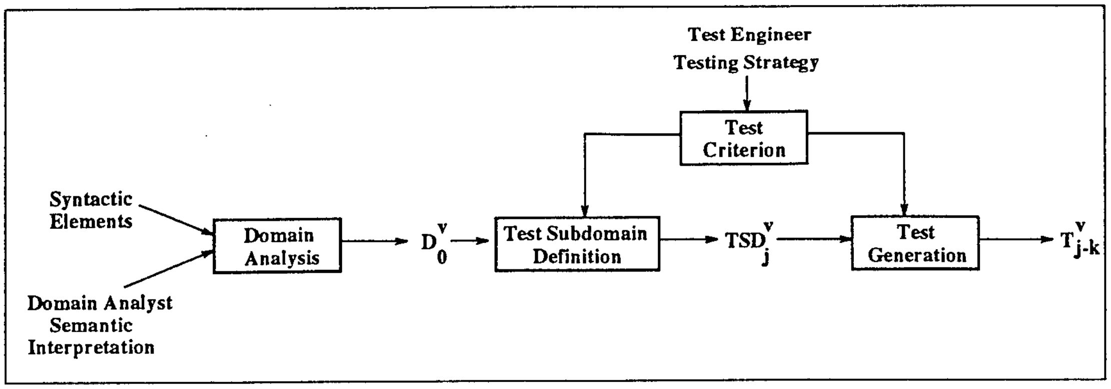


Figure 5.3: Detailed Test Generation Process Model

commands, parameters, and semantic rules applicable to the CAP are defined in the test subdomain.

Test criteria influence the test subdomain definition and the test generation steps. Test engineers use their knowledge to modify the domain model. They also guide test generation by recalling archived test suites, identifying how many commands to generate, and what commands to generate.

Test Generation takes information from the test subdomain and guidance from the tester (test criteria) to generate test suites, $T^v_{j-k}(k = 1, 2, 3, \ldots,)$. For instance, $T^{R1.2}_{CAP-10}$ denotes test #10 generated from the StorageTek CAP subdomain for Release 1.2. The remainder of the chapter shows how to translate test criteria into test subdomain modifications. This three stage test generation process also shows how information from the subdomain and the tester controls test generation. We conclude the chapter with an explanation of two test subdomains and a look at three test case reuse scenarios.

5.3 Test Subdomain Definition

Test subdomain definition customizes a domain model to focus test case generation. A test subdomain may be a subset or a superset of the original domain model. A subset restricts the parameters and commands generated in a test case and a superset allows greater freedom in test generation by turning semantic rules off (script rules, intracommand rules, parameter inheritance rules). Table 5.2 relates test criteria to test

Table 5.2: Relationship Between Test Subdomain Modifications and Test Criteria

Domain Model Component	Test Criteria	Test Subdomain Modifications
Script Class	Restrict Command Language Generate by operational profile	Turn Commands On/Off Adjust Command Frequencies
Script Rule	Test commands with no sequencing Force invalid command sequencing	Turn rules On/Off Modify Command Sequencing
Script Parameter Binding	Inconsistent parameter binding Force parameter binding faults	Turn Parameter Binding On/Off Modify Parameter Binding
Command Syntax	Test the parser Generate by operational profile	Modify Command Syntax Adjust Branch Frequencies
Pre/Post Conditions	Generate with no state info Force invalid states	Turn Off Pre/Post Conditions Modify Pre/Post Conditions
Intracommand Rule	Invalid parameter values Violate intracommand rules	Turn Intracommand Rules On/Off Modify Intracommand Rule
Parameter Value Set	Boundary-Value Tests Valid-Invalid Parameter Sets	Change Parameter Value Set
Parameter Constraint Rule	Random parameter values Test particular objects Test parameter constraint violations	Turn Parameter Constraints On/Off Modify Parameter Constraints

subdomain modifications. The table does not list all possible test criteria supported by Domain Based Testing. Instead, it suggests how a variety of test design strategies can be incorporated into the test generation process. In the next subsections, we detail each test criteria and its related test domain modification.

5.3.1 Test Subdomain Definition : Script Level

Scripts capture dynamic behaviors of the system under test using three components: Script Classes, Script Rules, and Script Parameter Binding. Scripting classes define sets of commands with similar functionality. Restricting the set of commands in a scripting class forces test generation to eliminate particular commands from the test case or focuses test generation on a subset of the application domain. For instance, tests run overnight should exclude commands that require human intervention. In addition to restricting commands in a scripting class, testers augment commands with relative frequency information. Data from operational profiles shows that some commands are generated more frequently than others. Using this information, test engineers develop realistic tests according to operational situations.

Table 5.3: Regular Expression Modification Operators

Expression	Meaning	Modification	Test Criteria
a	Generate a	$\neg a$	Do not generate a
a^+	Generate one or more a's	$\neg a$ a^*	Do not generate a Generate zero or more a's
a^*	Generate zero or more a's	NONE	Can't violate this rule

Table 5.4: Script Parameter Binding Modification Operators

Binding	Meaning	Modification	Meaning
p^*	Choose any valid value for p	p p^-	Choose previously bound value Choose any except previously bound value
p	Choose a previously bound value for p	p^* p^-	Choose any valid value for p Choose any except previously bound value
p^-	Choose any except a previously bound value for p	p^* p	Choose any valid value for p Choose previously bound value

Scripting Rules capture command sequencing information. Script rules can be turned off or they can be modified. Testers turn off sequencing rules when test criteria require "invalid" command sequences or when they test a command independently from its sequencing semantics. Testers modify script rules to alter command sequencing during test generation or to force "invalid" command sequences. Table 5.3 shows how to modify a script rule when represented as a regular expression. For instance, the regular expression `[Mount Any* Dismount]` generates the command `''Mount''` followed by zero or more commands from the *Any* class and it is terminated by the `''Dismount''` command. If a test criteria calls for violating all `Mount-Dismount` sequences, the modified regular expression `[¬Mount Any* Dismount]` meets the test design goal.

Script parameter bindings define parameter selection constraints between commands in a scripting rule. Test criteria may require modifications to these rules. For instance, testers generate random parameter selection by turning off parameter binding rules. Table 5.4 shows modifications to binding rules. Some modifications guarantee violations to the binding semantics while other modifications turn the binding rule into random parameter selection. Consider the StorageTek `Mount-Dismount` script rule with parameter binding annotations:

MOUNT *tape-id** *drive-id**
Any*
DISMOUNT *tape-id* *drive-id*

The binding shows the cartridge tape should be dismounted from the drive in which in was mounted. Suppose the test criteria calls for guaranteed parameter binding faults. The modified `Mount-Dismount` script rule below shows the parameter bindings to achieve this criteria:.

MOUNT *tape-id** *drive-id**
Any*
DISMOUNT *tape-id-* *drive-id-*

5.3.2 Test Subdomain Definition : Command Level

Domain information at the command level captures command syntax, pre/post conditions, and intracommand rules. Command syntax records productions of the command language using syntax diagrams or BNF. Testers modify, mutate, adjust or permute command syntax to test the command language parser. Such modifications include keyword errors, token misspellings, omitting tokens, or adding extra tokens. Testers augment command syntax with branch frequency information to model operational profiles, concentrate test generation on particular paths of a command, or eliminate certain paths from test generation. Information from an operational profile may show that some paths in the syntax of a command are generated more frequently than others. Testers configure the test subdomain to meet this objective by adjusting the relative frequency of a branch in a command's syntax.

Domain analysis associates pre/post conditions with each command. Preconditions describe the system state before a command can execute and postconditions define the system state after the command executes. Testers turn `Off` pre/post conditions when tests require no state information or when state information is not available. Suppose a test engineer needs to test a command or group of commands independent of the state of the system. DBT test generation records this criteria by turning off pre/post conditions. Pre/post condition modifications also provide a unique way to force command generation while violating system state rules. For instance, the StorageTek ENTER command requires a system state of:

```
service-level       = Full
lsm-status(lsm-id) = Online
lmu-status(lmu-id) = Online
```

Modifications listed below show how the test generator can be forced to create a sequence of commands that violate system state information associated with the ENTER command.

```
service-level       = Base
lsm-status(lsm-id) = Offline
lmu-status(lmu-id) = Offline
```

Intracommand rules capture parameter constraint rules within a single command. When defining a test subdomain, testers turn `Off` intracommand rules or they can modify them. If the rule is turned `Off`, the test generator makes no effort to enforce parameter constraints within the command. Parameter selection (within the command) becomes random. Test engineers modify intracommand rules by altering the first-order logic expression. For example, the intracommand rule for the MOVE command is: `(lsm$1=lsm$2) => (panel$1` $\neq$ `panel$2)`. When moving tapes within the same LSM, the source and destination panels must be different. Testers alter logic expressions like this by changing the relational operators. For instance, we can force the source and destination panels to be the same by changing the expression to: `if (lsm$1=lsm$2) => (panel$1 = panel$2)`.

5.3.3 Test Subdomain Definition : Parameter Value Selection

Test criteria also influence parameter value selection. Testers use parameter values to capture hardware and software configuration and parameter value constraints. Adjustments to the parameter values in a test subdomain provide opportunities to test valid, invalid, and boundary-value tests. Suppose parameter values are represented using mathematical sets. A *universal* set defines all possible values for a given parameter. A *default* set defines a subset of the universe. The default set typically specifies a particular configuration of the system under test. Testers define boundary-value, special configuration, and invalid parameter sets using set operations: `Union`, `Intersection`, and `Difference`. For example, test engineers use `invalid = universe - default` to define invalid parameter sets. Consider the StorageTek robot tape library. The universal set for the *lsm-id* parameter is `[000 ... FFF]`. The default set for their hardware test facility restricts this set to `[000 001 010]`. The invalid set is `invalid = universe - default = [002...00F, 011...FFF]`.

Test criteria may require violation of parameter constraint rules. A parameter constraint rule captures semantic information about the relationships between parameters. Typically, the value of one parameter constrains the possible values for another parameter. If these rules are turned `Off`, testers force the test generator to choose from `universal` or `default` parameter sets. Modifying parameter constraints may be needed when testing a particular object in the system or to test parts of the system that error check the hardware and software configuration. Parameter constraints can also indicate anomalous parameter situations. For instance, overconstraining parameter values may result in an empty set of choices. This could represent incorrect information in user documentation, "infeasible" or "unreachable" parameter value combinations for a system configuration, or incorrect capture of domain model information.

5.4 Test Generation

The *Test Generation* component of the Test Generation Process Model creates test suites, $T^v_{j-k}(k = 1, 2, 3, \ldots)$, based on test criteria from a test engineer and from a testing subdomain, TSD^v_j. Test generation follows a three staged sequence: *script expansion*, *template generation*, and *parameter value selection.* The three staged approach greatly simplifies the test generation process and reduces the number of active semantic rules at any one point. Spreading semantic rules across three stages avoids some of the complexity problems of other grammar-based approaches while making it possible to create a wide variety of test suites [DH81].

The first stage, Script Expansion, defines high-level tests by creating a list of command names. Testers provide guidance to the script stage by specifying what commands to generate, how many commands to generate. Command names are generated randomly and each name is examined for scripting rules. Tests archived at this stage are called *test scripts* and they can be recalled to re-generate a test, include the script in another test, or used during regression testing. In the second stage, each command name in the test script is expanded into a command template. We use the term template because parameter values are not selected. Place holders for the parameters are used instead. Tests archived at this stage are called *test templates.* In the last stage, the test generator creates a complete *test case* by replacing parameter place holders with actual values. This stage is the most complex because of several semantic rules that apply during parameter value selection. First, parameter binding for the scripting rule is checked. If the command marks the beginning of a scripting rule and the rule is turned on, then its parameters are pushed onto a stack. If the command marks the termination of a scripting rule, then its parameters are popped from the stack. While generating parameters for a single command, we also check for semantic rules at the command level. There may be many paths through a command's syntax so the intra-command rule may not apply to all command instances. Finally, individual parameters can be selected by following parameter constraint rules.

Table 5.5 lists functions required for the automated test generation process. Using this functions, one could implement a command-language, menu-driven, or GUI interface for DBT. The "Generate" commands are used in the scripting stage to create a list of command names by specifying a command or by choosing commands from a script class. The "save" and "include" functions used in all three stages allow the tester to archive and recall tests. The "merge" function shuffles two or more tests. This functions is used to simulate access to a shared device.

5.5 Example Test Subdomain Descriptions

In the next sections, we show two example test subdomains from the StorageTek Robot Tape Library. They demonstrate the flexibility of defining test subdomain according to a test criteria and from objects in the domain.

Table 5.5: Test Generator Functions

Stage	Function
Script	Generate N commands using <*command-name*>
	Generate N commands from Script Class <*class*>
	Save Test Script <*script-name*>
	Include Test Script <*script-name*>
	Merge <*script-name*> <*script-name*>$^+$
Command Template	Generate Command Templates
	Save Command Template <*template-name*>
	Include Command Template <*template-name*>
	Merge <*template-name*> <*template-name*>$^+$
Parameter Selection	Generate Parameter Values
	Save Test Case <*test-name*>
	Include Test Case <*test-name*>
	Merge <*test-name*> <*test-name*>$^+$

Table 5.6: Cartridge Access Port (CAP) Test Subdomain - $TSD_{CAP}^{R1.2}$

Domain Component	Test Subdomain Definition
Script Class (CAP-Class)	`CAPPREF DRAIN EJECT ENTER RELEASE SENTER`

Script Rules and Parameter Binding	`ENTER cap-id*`
	`<5/ANY>`
	`DRAIN cap-id`

Syntax	No Change
Pre/Post Conditions	No Change
Intracommand Rules	None

Parameter Value Sets	`host := {MVSE MVSH}`
	`acs := {00 01}`
	`lsm := {000 001 010}`
	`volser := {EVT180 EVT185 EVT199 EVT280 EVT289 EVT297 EVT393}`
	`subpool := {EVT1 EVT2}`
	`volcount := {1 3 10 15 20 39 47 53 62 77 85 94 100}`
	`cap := {000 001 010}`

Parameter Constraint Rules	`acs = 00 => lsm := {000 001}`
	`acs = 01 => lsm := {010}`
	`lsm = 000 => cap := {000}`
	`lsm = 001 => cap := {001}`
	`lsm = 010 => cap := {010}`

Table 5.7: MOVE Command Test Subdomain - $TSD_{Move}^{R1.2}$

Domain Component	Test Subdomain Definition

Script Class (Move-Class)
Script Rules and
Parameter Binding
	`EJECT ENTER DRAIN SENTER MOVE`
	`ENTER cap-id*`
	`<5/ANY>`
	`DRAIN cap-id`

Syntax	No Change
Pre/Post Conditions	No Change
Intracommand Rules	`(lsm$1=lsm$2) => (pp$1` $\neq$ `pp$2)`

Parameter Value Sets	`host := {MVSE MVSH}`
	`acs := {00 01}`
	`lsm := {000 001 010}`
	`volser := {EVT180 EVT185 EVT199 EVT280 EVT289 EVT297 EVT393}`
	`cap := {000 001 010}`
	`pp := {00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19}`
	`rr := {00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19}`
	`cc := {00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19}`

Parameter Constraint Rules	None - Choose from Default Parameter Value Sets

5.5.1 Cartridge Access Port (CAP) Test Subdomain - $TSD_{CAP}^{R1.2}$

Table 5.6 defines the Cartridge Access Port (CAP) Test Subdomain for Release 1.2 of the robot tape library command language. This test subdomain shows how to focus test generation on an object in the problem domain. All commands, parameters, and semantic rules associated with the CAP are defined. First, the `CAP-Class` captures a subset of the command language with commands that influence the CAP. The `Enter-Drain` command sequence is the only script rule defined in this subdomain. Command syntax as well as command pre/post conditions are not changed for this test subdomain. They will be used directly from $D_0^{R1.2}$. The parameter value sets define the `Default` choices for all parameters used by the `CAP-Class` commands. Parameter constraints show the restrictions placed on the default parameter set.

5.5.2 MOVE Command Test Subdomain - $TSD_{Move}^{R1.2}$

Table 5.7 defines the `Move` Test Subdomain for Release 1.2. This test subdomain shows how to focus test generation on a command from the command language. The

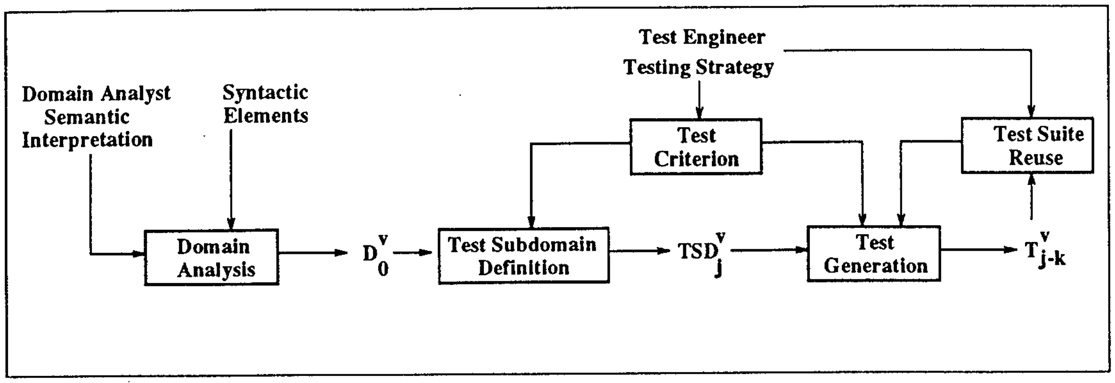


Figure 5.4: Test Generation Process Model with Reuse

`Move-Class` contains all commands needed to test tape movement. Commands to enter and eject tapes are included to test moving tapes in and out of an ACS. The `Enter-Drain` command sequence is the only script rule defined in this subdomain. Command syntax as well as command pre/post conditions are not changed. They will be used directly from $D_0^{R1.2}$. The parameter value sets define the `Default` choices for all parameters used by the `CAP-Class` commands. No parameter constraint rules will be used. Instead, the test generator chooses all parameter values randomly from the default sets.

5.6 Test Suite Reuse

While developing the Test Generation Process Model and experimenting with its capabilities, several test suite reuse ideas and scenarios emerged. Reuse is important because it saves test generation time, allows testers to test the "same things" uniformly, and it provides mechanisms to test different configurations the "same way."

Figure 5.4 shows how to incorporate test suite reuse into the Test Generation Process Model. Test suites, T_{j-k}^v, are recalled by test engineers. Test scripts and test templates must be re-generated using the test generation process because they are not fully parameterized lists of commands. Scripts and templates use information from the current test subdomain during re-generation. Fully parameterized test cases can be recalled without change or regeneration.

Table 5.8: Domain Based Testing - Reuse Applications

Test Generation Stage	Reuse Application	Reuse Category
Script Stage	Regression Testing	Software Version Reuse
	Command Syntax Change	Software Version Reuse
	New Software Release	Software Version Reuse
	Stress Test	Test Case Construction Reuse
	Creating new test scripts	Test Case Construction Reuse
	Operating System Version	Software Version Reuse
Command Template Generation	Regression Testing	Software Version Reuse
	Domain Model Change	Software Version Reuse
	Domain Test Subdomain Change	System Configuration Reuse
	Hardware Configuration Change	System Configuration Reuse
	Stress Test	Test Case Construction Reuse
	Creating new command templates	Software Version Reuse
	Parameter Value Change	System Configuration Reuse
Parameter Value Selection	Regression Testing	Software Version Reuse
	Re-run Test Case	Test Case Construction Reuse
	Creating new test cases	Test Case Construction Reuse
	Stress Test	Test Case Construction Reuse

Tests archived at all three stages of test generation offer many reuse applications (see Table 5.8). Each reuse application can be classified into one of three categories: Software Version Reuse, System Configuration Reuse, and Test Case Construction Reuse. Each category and its DBT reuse approach is described in the next sections.

5.6.1 Software Version Reuse

Software Version Reuse refers to test case reuse across software versions or releases. A command language interface may apply to different software versions or software releases. Each version contains slight differences in the command language to address variations in operating system, architecture, or system features. Consider a command language released for two operating systems. There may be a "core" set of commands that are similar, commands that only appear in a particular operating system version, or commands that have slightly different syntax because of OS or functional issues. In addition, the command language may change from one software release to the next. Such changes may add, delete, or modify the command language. Consider the line-print command for SunOS and HP-UX below. The commands have similar functionality by slightly different syntax.

```
SunOS : lpr -P[printer] [files...]
HP-UX : lpr -d[printer] [files...]
```

Software Version Reuse recalls archived tests at the *Scripting* stage. Test scripts contain command names (i.e., lpr in the line printer example above). From the script, a command template is generated using the current syntax from the test subdomain. From the test template, several fully parameterized test cases can be generated by varying the parameter value sets. Test generation from common test scripts builds uniform, comparable test suites for a variety of releases and platforms.

5.6.2 System Configuration Reuse

System Configuration Reuse refers to test case reuse across software and/or hardware system configurations. System configurations present a different reuse opportunity. A system under test may configure its logical and physical objects in a variety of ways. The domain model and test subdomain capture system configuration in parameter value files and parameter constraint rules. Consider the robot tape library configuration. An ACS supports from one to sixteen LSMs. LSM connections define the Pass-Thru-Ports of the system. Each LSM can vary its CAPs, Panels, Rows, Columns, and Tape Drives. A specific ACS may have one LSM while another has eight. Ideally, we would like to test both uniformly. We can do this via Software Configuration Reuse by recalling archived tests at the *Command Template* stage. The test generation process generates fully parameterized test cases from this template using the current parameter values set in the test subdomain. This scenario shows how to test different configurations the "same way."

5.6.3 Test Case Construction Reuse

Test Case Construction Reuse refers to reusing tests as "building blocks" for new tests. It employs reuse across all three test generation stages. Often testers find a particular list of commands good at detecting faults. Test engineers may also have system setup commands or workload generators to put the system in a particular state.

Consider the following. One script puts the system into a particular state. Another is added to present a workload to the system, and a third is included to test for a particular fault. The test generation process has mechanisms to combine command sequences like this into new test cases.

Test Case Construction is useful when leveraging pre-existing tests, creating stress tests, or testing a shared device. The DBT test generation process allows tests to include pre-existing test cases into a new test case. This provides an easy way to integrate DBT into a testing organization. A tester's existing work can be reused immediately. Sometimes, a tester needs to stress test a system. Using archived tests as building blocks, the tester has a variety of ways to build stress tests. For instance, a long sequence of commands may be needed to test a system over an extended period of time, or a long sequence may be needed to test a high command issue rate. In either case, sequences of archived tests can be connected to create larger tests. Finally, test case construction can benefit by *merging* tests. Merging allows the tester to "shuffle" several archived tests into a single test. This is important when testing shared devices because it provides a simple way to interleave commands from several users.

5.7 Summary

The Test Generation Process Model couples the domain model with a test generation process. Testers configure the domain model into a test subdomain based on test criteria. The test subdomain focuses test generation on a subset of the command language, objects in the domain, or test criteria for parameter value selection. Test engineers can modify all domain model components. The three staged test generation sequence uses information from the test subdomain and the test engineer to create test suites. The scripting stage generates a sequence of command names. The command template stage creates an instance of the command with place holders for parameters. The last stage selects parameter values for the command template. Test generation is simplified by spreading semantic rules across all three stages. Tests can be archived at all three test generation stages. This provides an opportunity to reuse tests for a variety of applications. Tests can be recalled to test different versions of a command language, test different system configurations, or to construct new tests.

Chapter 6

TEST GENERATOR IMPLEMENTATION

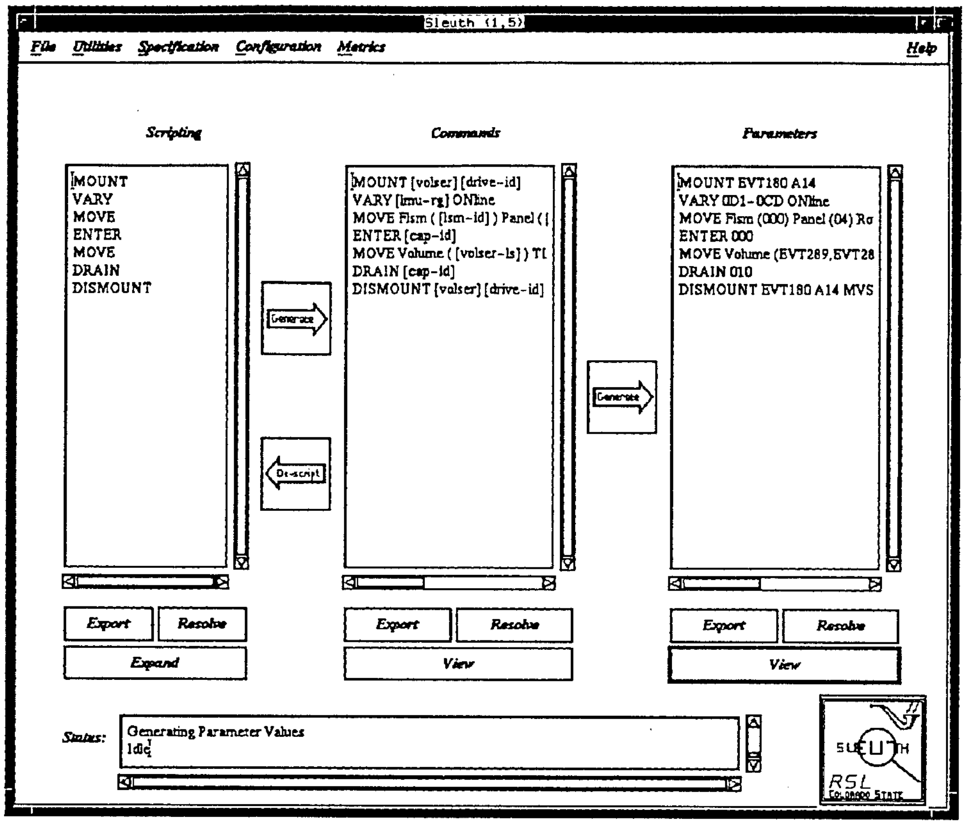
6.1 Introduction

This chapter presents and compares two DBT test generators. *Sleuth* uses a hybrid collection of sentence generation algorithms and tools [Wal94]. The second uses an AI Planner called UCPOP. Both test generators follow the test generation process as described in the previous chapter. *Sleuth* supplies a set of tools to define domain models, configure test subdomains, generate tests, and archive tests. *Sleuth* maintains a complete domain model for the HSC Release 1.2 command language. The AI Planner is an experimental test generator for DBT. We used it to explore alternatives for DBT test generation. The planner uses a subset of the HSC domain model for its test generation. The subset or experimental subdomain contains all domain model features but it does not contain the entire HSC command language.

6.2 Hybrid Implementation

Sleuth is an automated test generation tool developed at Colorado State University. *Sleuth* supports Domain Based Testing by providing tools and utilities for test generation. The graphical user interface (GUI) was programmed using Motif [Bra92, O'r93]. The main window models the test engineers' test generation process (see Figure 6.1). The menu bar provides several utilities to create domain models (specification), test subdomains (configuration), and to generate test cases.

Sleuth uses a hybrid collection of sentence generation tools and utilities. We use a hybrid approach because it was easy to iteratively develop and improve the test generator. As we refined DBT and its test generation process, we could substitute or



```
MOUNT
VARY
MOVE
ENTER
MOVE
DRAIN
DISMOUNT
```

```
MOUNT [volser] [drive-id]
VARY [lmu-rg] ONline
MOVE Flsm ( [lsm-id] ) Panel ( [
ENTER [cap-id]
MOVE Volume ( [volser-ls] ) T(
DRAIN [cap-id]
DISMOUNT [volser] [drive-id]
```

```
MOUNT EVT180 A14
VARY 0D1-0CD ONline
MOVE Flsm (000) Panel (04) Ro
ENTER 000
MOVE Volume (EVT289,EVT28
DRAIN 010
DISMOUNT EVT180 A14 MVS
```

```
Generating Parameter Values
Idle
```

Figure 6.1: *Sleuth* Main Window - Three Stages of Test Generation

experiment with different tools. Table 6.1 lists the domain model components and the hybrid implementation. Using this implementation, we define a domain model for the entire StorageTek HSC Release. 1.2 command language [1]. We will use parts of it to illustrate *Sleuth.*

6.2.1 Script Representation

Scripts capture dynamic behavior of the system under test. *Sleuth* stores Script Classes, Script Rules, and Script Parameter Binding. A scripting class is a set of command names with similar functionality. The notion of a scripting class helps testers select *what type* of commands should be generated for a test case. The number of

[1]See Appendix 1 for the complete domain model.

Table 6.1: Domain Model Components and Hybrid Implementation

Domain Model Component	Hybrid Implementation
Script Classes	Sets of Command Names
Script Sequencing Rules	Macro Expansion
Script Parameter Binding	Macro Expansion
Command Language Syntax	Syntax Diagrams
Command Preconditions	Implicit representation
Command Postconditions	Implicit representation
Intracommand Rules	First Order Logic
Parameter Constraint Rules	Parameter Value Sets Parameter Constraints based on Set Operations

Table 6.2: Script Classes for the **StorageTek** HSC Domain

Script Class	Commands					
Any	Alloc	Commpath	Eject	Mntd	Move	Retry	Srvlev
	Cappref	Dismount	Enter	Modify	Option	Scrparm	Switch
	Cds	Display	Journal	Monitor	Recover	Senter	Trace
	Clean	Drain	Load	Mount	Release	Set	Uexit
Mode	Cappref	Clean	Mntd	Option	Set	Trace	Warn
	Cds	Journal	Monitor	Scrparm	Stopmn	Uexit	
Set-Up	Alloc	Journal	Option	Srvlev	Trace	Cappref	Mntd
	Scrparm	Stopmn	Uexit	Commpath	Modify	Set	Switch
	Vary						
Action	Alloc	Display	Enter	Move	Retry	Commpath	Drain
	Load	Recover	Senter	Dismount	Eject	Mount	Release
	View						

classes and the types of script classes is problem dependent. The script classes for the StorageTek domain are listed in Table 6.2. They are assigned to four script classes. The Any class is the *universal* set and it contains all commands in the HSC command language. The Mode class contains commands that modify the operating mode of the ACS. Set-up commands perform machine set-up. Action commands cause physical actions within the ACS.

Script Rules define command sequencing and capture system state information during test generation. In the hybrid implementation, macro expansion ensures that commands are sequenced properly. A macro expansion is a sentence generation mechanism that replaces one string for another. Each macro is given a name and during processing each instance of the macro name is is replaced by another string. Macros can also be parameterized. For DBT, we use a simple string replacement macro expansion mechanism. Consider the robot tape library. A script rule states that one cannot "dismount" a tape from a tape drive unless one has previously been "mounted." The macro representation for this rule is:

```
MOUNT <tape-id*> <volser*>
<5/any>
DISMOUNT <tape-id> <volser>
```

Let's examine the macro expansion procedure in more detail. During the scripting stage, commands are randomly selected. If a MOUNT or DISMOUNT command is generated, the macro expansion rule creates a MOUNT command followed by up to five commands from the Any class. This sequence is terminated by a DISMOUNT command. *Parameter binding* makes sure the parameters in the command sequence are meaningful. For instance, parameter binding information ensure that the tape that is "mounted" is the same tape that is "dismounted."

Script rules also represent state information maintained during test generation. State information is required to restrict commands from being generated, forcing the test generator to choose a particular command, and to capture parameter bindings between commands in a test case. To show how a script rule represents state information, one can draw a state transition diagram based on the macro expansion rule. Given a state in the diagram, the state transitions (i.e., arcs) define what commands can be issued.

6.2.2 Command Representation

The second component of the domain model captures the syntax and semantic rules for each command. Command syntax is represented with syntax diagrams. A random traversal through the diagram during test generation creates an instance of a command. *Sleuth* enhances each syntax diagram with branch frequency data. The frequency information alters the chances of branch selection during the traversal. Branch frequencies set to zero effectively eliminate the branch from being taken. Figure 6.2 shows the ENTER command as represented by the *Sleuth* Syntax Diagram Editor.

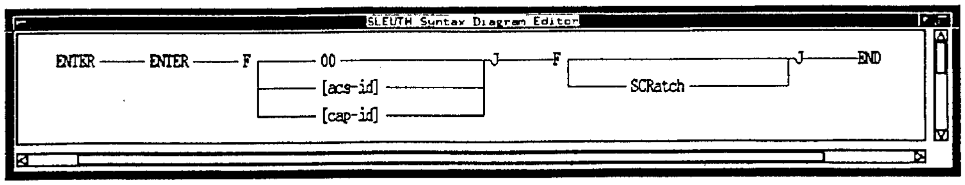


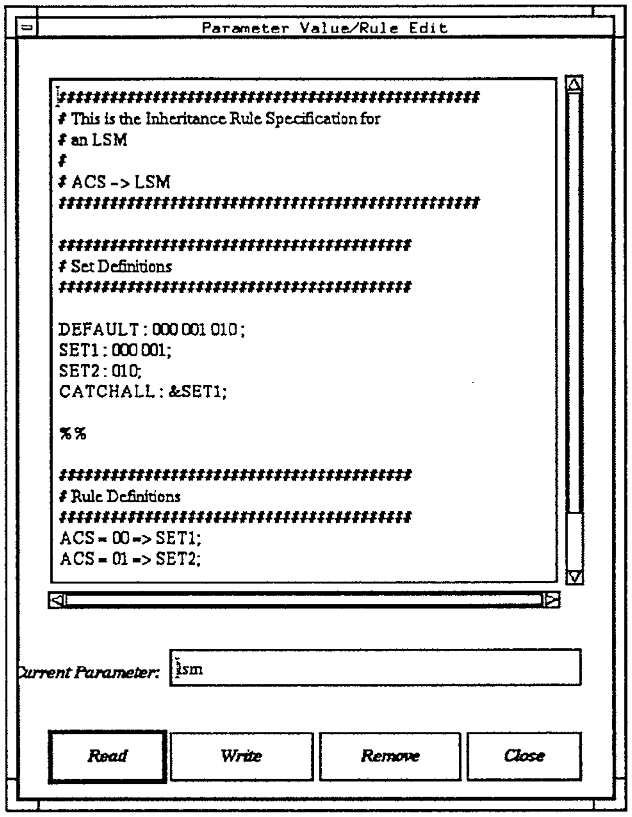
Figure 6.2: Syntax Diagram Editor

Three types of semantics rules are defined at the command level: *preconditions*, *postconditions*, and *intracommand rules.* Preconditions identify the conditions that must hold before the command can execute. Postconditions list the conditions that are true after the command executes. Preconditions and postconditions are implicitly represented in

Intracommand rules identify constraints placed on parameter value selection within a single command. For example, StorageTek's automated tape library requires that if moving tapes within the same LSM, then the source and destination panels must be different. This is represented in the domain model in first-order logic : `(lsm$1=lsm$2) => (panel$1` $\neq$ `panel$2)`. During test case generation, this rule may or may not be applicable when choosing parameters for the `MOVE` command. First, parameter values for `lsm$1` and `lsm$2` must be selected and they must be equal for the rule to apply. Second, there are many paths through a command's syntax. Some of the paths may not require the application of the intracommand rule.

6.2.3 Parameter Representation

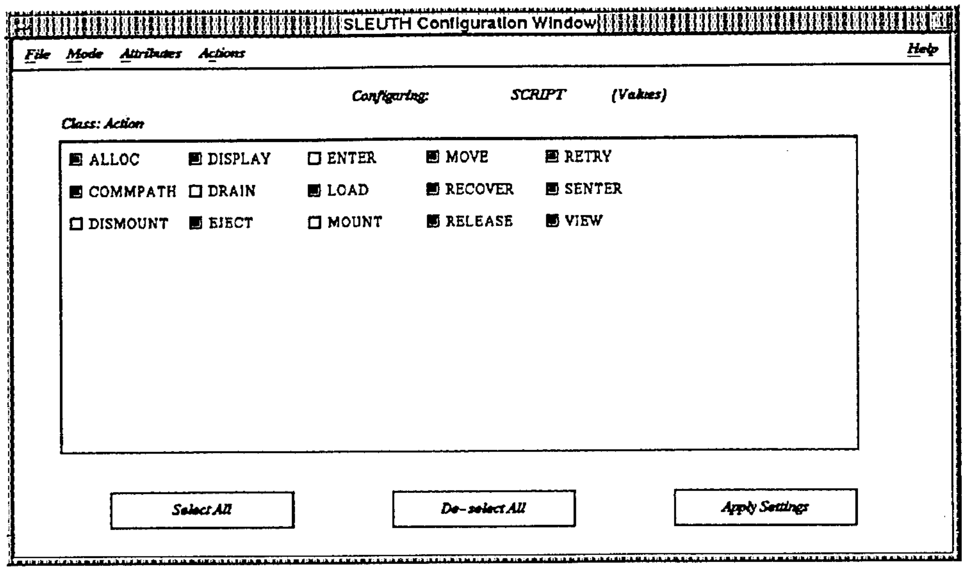
Parameters of the command language are represented as Parameter Value Sets. Using set operations (Union, Intersection, and Difference) a variety of parameter sets can be defined. During test generation, parameter values may be constrained. For example, the value for a particular LSM (*lsm-id*) constrains possible values for panels, rows, and columns for cartridge storage. These constraints are defined because the physical layout of an LSM can vary. Parameter constraint rules are represented using a "parameter" hierarchy that denotes relationships between "parameters". Set operations



```
##################################################
# This is the Inheritance Rule Specification for
# an LSM
#
# ACS -> LSM
##################################################

##########################################
# Set Definitions
##########################################

DEFAULT : 000 001 010 ;
SET1 : 000 001;
SET2 : 010;
CATCHALL : &SET1;

%%

##########################################
# Rule Definitions
##########################################
ACS = 00 => SET1;
ACS = 01 => SET2;
```

lsm

Figure 6.3: Parameter Value Editor



Configuring: SCRIPT (Values)

Class: Action

■ ALLOC	■ DISPLAY	□ ENTER	■ MOVE	■ RETRY
■ COMMPATH	□ DRAIN	■ LOAD	■ RECOVER	■ SENTER
□ DISMOUNT	■ EJECT	□ MOUNT	■ RELEASE	■ VIEW

Figure 6.4: The Configuration Module (Dark buttons = Command is `On`)

modify parameter value sets to restrict choices during parameter selection. Figure 6.3 shows how the *Sleuth* parameter editor represents parameter value sets and parameter constraint rules for the HSC *lsm* parameter.

6.2.4 Test Subdomain Configuration

Test engineers define *Test Subdomains* using *Sleuth*'s `Configuration` pull-down menu. Test subdomains can be saved and loaded. Some of the features available to the tester include:

- Turn commands within a scripting class On or Off
- Turn command rules On or Off
- Modify the frequency of each command
- Modify the Syntax Diagram for each command
- Modify parameter values/rules
- Modify scripting and command rules

To illustrate, Figure 6.4 shows a screen where commands from the `Action` scripting class can be turned `on/off`. This has been particularly useful to test the tape library overnight since testers can turn commands `off` that require operator intervention.

Table 6.3: *Sleuth* Implementation of the Test Generator Functions

Stage	Function	*Sleuth* Implementation
Script	Generate N commands using <*command-name*>.	@n\<*command-name*> Press Expand Button
	Generate N commands from Script Class <*class-name*>.	@n\<*class-name*> Press Expand Button
	Save Test Script <*script-name*>.	Press EXPORT Button Enter <*script-name*>
	Recall Test Script <*script-name*>.	%include <*script-name*> Press Resolve Button
Command Template	Generate Command Templates	Press Generate Button
	Save Command Template <*template-name*>.	Press EXPORT Button Enter <*template-name*>
	Recall Command Template <*template-name*>.	%include <*template-name*> Press Resolve Button
Parameter Selection	Generate Parameter Values	Press Generate Button
	Save Test Case <*test-name*>.	Press EXPORT Button Enter <*test-name*>
	Recall Test Case <*test-name*>.	%include <*test-name*> Press Resolve Button

6.2.5 Example Test Generation

Table 6.3 relates test generator functions defined in Chapter 5 with the *Sleuth* implementation. Some functions require typing a command into one of the *Sleuth* windows. Others are executed with a simple button press. This section demonstrates a typical test generation sequence.

Table 6.4 shows an example of a test generation request for ten commands from the Any class. The command [@20/Any] is typed into the Script window on the *Sleuth* main window. Then, the tester presses the Expand button. The resulting list of commands is displayed in the second column in Table 6.4. More than twenty commands are generated because some commands require sequencing. A randomly generated command name may expand (i.e., macro expand) into a sequence of commands. This results in more commands than originally requested. Figure 6.5 shows three passes of the script expansion algorithm. Each command with an active script rule requests *up to* five commands from the Any class. Expanding a command into sequence of commands can be recursive. A single request could create a long list of command names or the test gener-

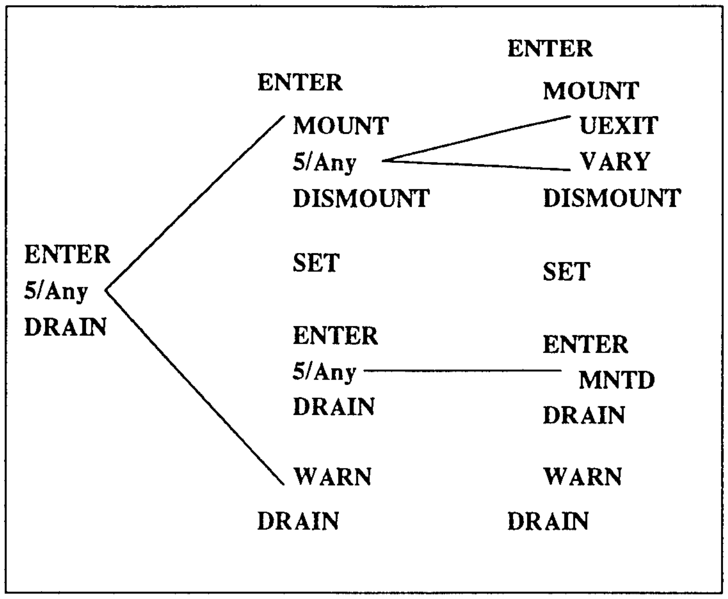


Figure 6.5: Script Expansion Example

ator could fail because of memory limitations. This problem is short-circuited in *Sleuth* by specifying a maximum number of recursive calls. This limit is user specified.

Table 6.5 shows the second stage of test generation. Testers generate command templates by pressing the `Generate` button in the *Sleuth* main window. The test generator expands the list of command names into command templates. Parameter place-holders are denoted in square brackets. Each template is generated by taking a random traversal through each command's syntax digram. Table 6.6 shows the final test case. Testers generate a complete test case by pressing the `Generate` button on the *Sleuth* main window. The test is a list of fully parameterized commands. Commands obey script rules and script parameter binding. For instance, each `Mount-Dismount` pair matches tape identifier and tape drive parameters.

Once *Sleuth* completes a test case, the tester can request *test case metrics* that may be helpful to a test engineer during test case construction. The metrics are calculated with respect to domain model components. For example, *branch coverage* refers to the branches in the grammar of the command. *Node coverage* measures the number of nodes (i.e., terminal or nonterminal in the grammar) that were exercised by the test case. We

Table 6.4: Script Generation Example

Generation Request	Expanded Script
@20/Any	ENTER MOUNT UEXIT VARY DISMOUNT SET ENTER MNTD DRAIN WARN DRAIN CAPPREF DISPLAY ALLOC MNTD ENTER SCRPARM MOUNT LOAD WARN DISMOUNT

Table 6.5: Command Template Example

Script	Command Template
ENTER	ENTER 00
MOUNT	MOUNT [volser] [drive] [host-id] Readonly
UEXIT	UEXIT ([uexit-ls]) Query
VARY	VARY [lmu-id] OFFline FORCE
DISMOUNT	DISMOUNT [volser-id] [drive-id]
SET	SET ENTdup Manual
ENTER	ENTER 00
MNTD	MNTD MOuntmsg(Noroll), MAXclean(100), Dismount(Manual)
DRAIN	DRAIN [cap-id]
WARN	WARN SCRatch [lsm-id] THREShld([warn-thresh])
DRAIN	DRAIN [cap-id]
CAPPREF	CAPPREF [prefvlue] 000 [host-id]
DISPLAY	DISPLAY Volume [volser-rg] DEtail
ALLOC	ALLOC Unitaff (NOSep) HOSTID ([host-id])
MNTD	MNTD AUtocln (ON)
ENTER	ENTER 00 SCRatch
SCRPARM	SCRPARM [initwarn] 4 [secwarn] 4 1
MOUNT	MOUNT [volser] [drive]
LOAD	LOAD SLSLDQR
WARN	WARN SCRatch [acs-id] SUBpool([subpool]) THREShld([warn-thresh])
DISMOUNT	DISMOUNT , [drive-id]

Table 6.6: Test Case Example

Test Case
ENTER 000
MOUNT EVT289 A29 MVSE Readonly
UEXIT (03,02,08,04,13) Query
VARY 0CE OFFline FORCE
DISMOUNT EVT289 A29
SET ENTdup Manual
ENTER 000
MNTD MOuntmsg(Noroll), MAXclean(100), Dismount(Manual)
DRAIN 000
WARN SCRatch 000 THRESHld(0)
DRAIN 000
CAPPREF 6 000 MVSE
DISPLAY Volume EVT280-EVT289 DEtail
ALLOC Unitaff (NOSep) HOSTID (MVSE)
MNTD AUtocln (ON)
ENTER 00 SCRatch
SCRPARM 04 4 20 4 1
MOUNT EVT280 A14
LOAD SLSLDQR
WARN SCRatch 00 SUBpool(EVT1) THRESHld(333)
DISMOUNT , A14

use these measures as input to a neural network classifier in Chapter 7. Figure 6.7 lists node and branch coverage for the sample test case in Figure 6.6.

6.3 AI Planner Implementation

We also experimented with using an AI Planner as an alternative DBT test generation "engine." This prototype demonstrated the possibilities of using a planner for automated test generation. These experiments used a subset of the StorageTek HSC command language. The subset or *experimental subdomain* includes all domain model components. Table 6.8 lists each domain model component and its AI planning representation.

We implemented the experimental subdomain in the UCPOP planner [BGPW93]. The planner was selected because it is easy to use and the software easily obtained. UCPOP is a "Universal Conditional Partial Order Planner" which means that it can represent goals that include universal quantifiers (e.g., move *all* tapes) and that it does not order the sequence of operators in the plan until necessary, which makes it more

Table 6.7: Node and Branch Coverage for Figure 6.6

Command	Number Nodes	Node Coverage	Number Branches	Branch Coverage
ALLOC	32	16%	23	17%
CAPPREF	7	57%	6	33%
DISMOUNT	6	67%	4	75%
DISPLAY	60	7%	63	6%
DRAIN	2	100%	0	0%
ENTER	6	50%	5	60%
LOAD	3	67%	2	50%
MNTD	41	22%	25	40%
MOUNT	14	36%	14	29%
SCRPARM	11	55%	12	50%
SET	32	9%	36	6%
UEXIT	19	16%	12	17%
VARY	7	57%	7	43%
WARN	12	67%	4	100%

Table 6.8: Domain Model Components and AI Planner Representation

Domain Model Component	Planner Implementation
Script Classes	Collection of Planner Operators
Script Sequencing Rules	Operator preconditions
Script Parameter Binding	Operator effects
Command Language Syntax	Operators : One operator for each "path" in a command. Postprocessor : Translates Planner output into Command Language Syntax.
Command Preconditions	Operator preconditions
Command Postconditions	Operator effects
Intracommand Rules	Operator preconditions
Parameter Constraint Rules	Preprocessor : Initial State Generator Preprocessor : Goal Generator

flexible. The planner requires two preprocessors, a set of planner operators, and a postprocessor. The preprocessors generate an initial state and a goal state for the planner. The operators describe the commands of the command language, and the postprocessor translates the planner output into the correct StorageTek syntax.

6.3.1 Script Representation

A script class is a set of commands with similar functionality. Classes are represented as collections of planner operators. Testers guide test case generation by loading or excluding these collections during the planner's initialization. For instance, the ex-

```
;;Description : Mount a tape in a tape drive.
;;Precondition : Service-Level = FULL.
;;               Tape is inside the LSM.
;;               LSM-status = ONLINE.
;;Postcondition: Tape is in the tape drive.
(define   (operator mount)
          :parameters ((loc ?slsm) ?vid ?m_did ?p ?c ?r )
          :precondition (and (full slev)(in ?vid ?slsm ?p ?r ?c)(on ?slsm))
          :effect (and (at ?vid ?m_did)(not(in ?vid ?slsm ?p ?r ?c))))

;;Description : Dismount a tape from a tape drive.
;;Precondition : Service-Level FULL
;;               Tape is in the tape drive.
;;Postcondition: Tape is placed back into the LSM.
(define   (operator dismount)
          :parameters(?vid ?m_did ?d_did ?p ?c ?r)
          :precondition(and (full slev)(at ?vid ?m_did)(eq ?m_did ?d_did))
          :effect (and (not (at ?vid ?m_did))(backtolsm ?vid through ?d_did)
                       (in ?vid unknown unknown unknown unknown)))
```

Figure 6.6: Planning representations of the Mount and Dismount commands

perimental subdomain focused on "moving" tapes within the tape library. By including or excluding certain "move" operators, we could change test case generation, produce different command sequences for similar goals, or focus on specific types of tape movement.

Script sequencing rules and parameter binding make sure commands are issued in the correct order. The AI planning representation includes this information in the preconditions and effects of the planner operators. The `MOUNT-DISMOUNT` sequencing rule is implemented with the planner operators listed in Figure 6.6. Preconditions for the `DISMOUNT` require a tape to be loaded in a tape drive. A tape can be placed into a drive in one of two ways: (1) The tape drive can be loaded as part of the initial state, or (2) the tape can be loaded as an *effect* of the `MOUNT` operator.

6.3.2 Command Representation

The next domain model component is command language syntax. The AI planner does not explicitly use the syntax of the command language. Instead, each path through a command is represented as a separate planning operator. A postprocessor translates the planner output into the correct syntax. For instance, the StorageTek

```
;;Description  : Change Service-Level to FULL.
;;Precondition : None.
;;Postcondition: Service-Level = FULL.
(define   (operator servicetofull)
          :precondition (base slev)
          :effect (and(not(base slev))(full slev)))

;;Description : Change Service-Level to BASE.
;;Precondition : None.
;;Postcondition: Service-Level = BASE.
(define   (operator servicetobase)
          :precondition (full slev)
          :effect (and (not (full slev)) (base slev)))
```

Figure 6.7: Example planning representations of the command language syntax: changing service level

command language uses the `SRVLEV` command to "toggle" the system's service level: `service-level-cmd ::= SRVLEV {BASE | FULL}`. As seen in Figure 6.7, two operators encode the command for the planner.

The next three domain model components are command preconditions, command postconditions, and command intracommand rules. All three are represented as preconditions and effects to planner operators. A command language precondition denotes sequencing information based on system state. If the system is not in the correct state, the "precondition" provides information to put the system in the proper state. Likewise, command language postconditions specify how the state of the system changes upon executing an operator. Intracommand rules are also specified as preconditions to planner operators. These preconditions check parameter values within the command. Figure 6.8 shows one version of the `MOVE` command. This operator encodes the intracommand rule: `(lsm$1=lsm$2) => (panel$1` $\neq$ `panel$2)` by requiring `slsm` (source LSM) and `dlsm` to be equal (`eq`) and `sp` (source panel) and `dp` (destination panel) to be not equal (`neq`).

6.3.3 Parameter Representation

The last domain model component represents command language parameters and parameter constraints. The AI planner uses two preprocessors to capture this informa-

```
;;Description : Move a volume in a specific location to the destination LSM,panel.
;;Precondition : Source and destination LSM are online.
;;               Service level is full.
;;               Volume location is specified by LSM, panel,row,column.
;;               Source and destination LSMs are connected.
;;Intracommand : Source and destination LSMs must be equal.
;;               Source and destination panels must be different.
;;Postcondition: Move the volume to a new panel inside the same LSM.

(define  (operator movefour)
         :parameters ((loc ?slsm) ?sp ?sc ?sr (loc ?dlsm) ?dp ?dc ?dr (tape ?vid))
         :precondition (and (full slev)(on ?slsm)(on ?dlsm)
                            (eq ?slsm ?dlsm) (neq ?sp ?dp)
                            (in ?vid ?slsm ?sp ?sr ?sc)
                            (eq ?dc unknown) (eq ?dr unknown))
         :effect (and (from ?vid ?slsm ?sp ?sr ?sc ?dlsm ?dp ?dr ?dc)
                      (in ?vid ?dlsm ?dp ?dr ?dc)
                      (not (in ?vid ?slsm ?sp ?sr ?sc))))
```

Figure 6.8: Example planning representation of the command preconditions, postconditions and intracommand rules: move a volume

```
initial-state =    ((BASE SLEV) (LOC 0) (ON 0) (CAP 0 ENTERING) (LOC 1) (OFF 1)
                   (CAP 1 ENTERING) (LOC 10) (OFF 10) (CAP 10 ENTERING)
                   (CONNECT 0 1) (TAPE EVT297)(IN EVT297 10 UNKNOWN
                   UNKNOWN UNKNOWN))

goal =             (AND (FROM EVT280 0 UNKNOWN UNKNOWN UNKNOWN 1
                         UNKNOWN UNKNOWN UNKNOWN)
                         (FROM UNKNOWN 0 1 2 3 0 4 5 6))
```

Figure 6.9: Example planning representation of objects, object elements and parameter constraints: initial state list and goal list

tion: Initial State Generator and Goal Generator. The initial state generator randomly creates an initial state vector for the planning system. This state vector uses information about parameter constraints to make sure the state is valid. In the StorageTek experimental subdomain, we concentrated on two test generation goals: Moving tapes and Mounting/Dismounting tapes. Therefore, all state information necessary for these experiments was included in the state vector. Figure 6.9 shows an example of an initial state.

The second preprocessor uses object and parameter constraint information to generate a goal for the planner. A single goal for these experiments is to move an individual tape or to mount-dismount a single tape. If more than one move or mount-dismount is needed, a conjunctive goal is created. An example of a compound goal is shown in Figure 6.9. The fields in the goal statement are:

```
(FROM [tape-id] [src  lsm] [src  panel] [src  row] [src  column]
                [dest lsm] [dest panel] [dest row] [dest column])
```

The first subgoal requests the system to move tape EVT280 from LSM 000 to LSM 001. In this goal we are not concerned about where the tape is located. We are only concerned about moving it to a different LSM. In the second subgoal, a tape located in panel 1, row 2, and column 3 is moved within the same LSM to panel 4, row 5, and column 6. These two examples show how testers can focus test generation at different levels of abstraction through planning system goals.

6.3.4 Test Subdomain Definition

The AI planning system is able to configure a test subdomain similar to the hybrid implementation. By "configure", we mean that test subdomains can be defined by generating constrained (e.g., illegal) or restricted (e.g., just use one type of command) types of tests. For instance, script rules and script parameter binding can be altered by removing ("mutilating") precondition lists and by changing relational operators, respectively. Intracommand rules can be eliminated by removing parameter binding preconditions or by altering relational operators. Parameter value selection can be adjusted by modifying the preprocessors. Finally, command syntax can be altered by changing the postprocessor.

6.3.5 Example Test Generation

Figure 6.10 shows one run of the UCPOP Planner given the goal of moving a tape from one LSM to another. The initial state shows that the tape is located in LSM 000 (`(IN EVT280 0 UNKNOWN UNKNOWN UNKNOWN)`) and both the source and destination LSMs are offline (`(OFF 0) (OFF 1)`). In the StorageTek system, it may not be possible to know all of the state information for particular entities. This is indicated with the `UNKNOWN`

Initial:	((FULL SLEV) (LOC 0) (OFF 0) (CAP 0 DRAINED) (LOC 1) (OFF 1) (CAP 1 ENTERING) (LOC 10) (ON 10) (CAP 10 DRAINED) (CONNECT 0 1) (LOC ACS) (TAPE EVT280) (IN EVT280 0 UNKNOWN UNKNOWN UNKNOWN))
Goal:	(FROM EVT280 0 UNKNOWN UNKNOWN UNKNOWN 1 UNKNOWN UNKNOWN UNKNOWN)
UCPOP:	
Step 1:	(MODIFYTOON 0) Created 3 0 -> (FULL SLEV) 0 -> (OFF 0)
Step 2:	(MODIFYTOON 1) Created 2 0 -> (FULL SLEV) 0 -> (OFF 1)
Step 3:	(MOVEONE EVT280 1 0 UNKNOWN UNKNOWN UNKNOWN UNKNOWN UNKNOWN UNKNOWN) Created 1 3 -> (ON 0) 2 -> (ON 1) 0 -> (FULL SLEV) 0 -> (CONNECT 0 1) 0 -> (IN EVT280 0 UNKNOWN UNKNOWN UNKNOWN) 0 -> (TAPE EVT280) 0 -> (LOC 1) 0 -> (LOC 0)
Postprocessor:	
step 1 is:	(MODIFY 000 ONLINE)
step 2 is:	(MODIFY 001 ONLINE)
step 3 is:	(MOVE VOLUME(EVT280) TLSM(001))

Figure 6.10: Example Results from UCPOP : Goal = Move Tape EVT280 from LSM 000 to LSM 001

place holder. UCPOP's solution is listed in steps 1-3. In steps 1 and 2, the planner turns LSMs 000 and 001 `Online`. In step 3, the planner achieves the goal of moving the tape from one LSM to another. The Postprocessor translates the UCPOP output into the correct HSC syntax.

6.4 Comparison

At the most basic level, we were able to use the AI planning system to generate a variety of test cases in the experimental subdomain similar to tests generated by *Sleuth*. UCPOP required little "code" to represent the experimental subdomain. The two preprocessors, one postprocessor, and 18 operators needed 414 lines of code.

For *Sleuth*, the entire test generation tool required about 25,000 lines of C (internals and Motif interface). However, *Sleuth* implemented a much larger test domain than UCPOP and included a sophisticated user interface. Despite the small code size in the planner, computation time for a test case could take from minutes to hours. This was not a surprise since the planner was developed as a prototype and no efforts were made to improve its efficiency. Additionally, UCPOP was designed for ease-of-use and theoretical soundness. We compared the two implementations in two ways: whether the test cases covered similar aspects of the domain and what kinds of test cases were generated by the planning system.

6.4.1 Domain Coverage Comparison

We compared the domain coverage of the planning and hybrid (i.e., *Sleuth*) implementations by considering how each represented the seven levels of the domain model (as appear in Table 6.1). Both the hybrid and planner representations employed similar mechanisms to implement Script Classes. The hybrid representation stores command names in a set, and the planner includes or excludes operators to form script classes.

Scripting Rules (sequencing rules and parameter bindings) require different approaches in the two representations. The hybrid scheme needs command sequencing information in the first stage of test generation but it does not need parameter binding information until the last stage. The AI planner includes sequencing information and parameter binding rules in each operator.

The Command Syntax is represented differently in the two test generation engines. The planning system encodes each "path" through a command as a separate operator. One operator could represent multiple paths so long as all paths use the same parameters, the same preconditions, and produced the same effects; UCPOP allows some flexibility in the preconditions but little in the definition of the effects. The final test case is generated using a postprocessor to translate the UCPOP output into the appropriate StorageTek syntax. The hybrid approach stores each command as a syntax diagram. *Sleuth* takes a random traversal through the syntax diagram to create a command template.

Command Preconditions and Postconditions are not explicitly represented in the hybrid representation. Instead, testers must issue setup commands or execute a list of commands to put the system in the correct state or the tester defines script rules that expand into a sequence of commands placing the system in a desired state. The planner explicitly represents pre/post conditions in each planning system operator. The tradeoff between these two representations is in the amount of state information required during test generation. The planner requires much more specification; it does not tolerate ambiguity, nor incomplete information. *Sleuth*, on the other hand, does not require a full specification and it can generate tests with or without complete system information.

Parameter value sets and parameter constraints are handled differently in both domain model representations. The hybrid approach uses parameter value files, set definitions, and set operations. The default values for a parameter are defined using sets and set operations. If one parameter value constrains the choices for another, additional set operations allow *Sleuth* to choose from a constrained set. In the planning system, all parameter information was encoded into two preprocessors: Initial State Generator and Goal Generator. Both use information about default parameter values and parameter constraint rules to define initial states and goals that do not violate the parameter selection rules.

6.4.2 Nature of Tests

One of the most interesting aspects of these results were the differences between the planning approach and the hybrid approach to test data generation. Testers take different views of the problem during test generation. Using the hybrid approach, the tester focuses on what subset of commands to generate and how many commands. Using the planner, the tester describes the desired *outcome* and allows the planner to choose the appropriate sequence of commands to achieve the goal. We view planner based (goal oriented) test generation as a natural way to generate tests.

To demonstrate the differences consider the test in Figure 6.9. The objective was to move a single tape. *Sleuth* was directed to randomly generate four commands; the test

Table 6.9: Comparing *Sleuth* and UCPOP Tests

	Sleuth Test Case	UCPOP Test Case
1	`MODIFY 001 ONline`	`MODIFY 000 ONLINE`
2	`MODIFY 000 ONline`	`SRVLEV FULL`
3	`MODIFY 000 ONline`	`DRAIN 000`
4	`ENTER 000`	`ENTER 000`
5	`MOVE (EVT289) Tlsm(000)`	`MOVE VOLUME(EVT280) Tlsm(001)`
6	`MODIFY 001 ONline`	
7	`DRAIN 000`	

case was selected when it included a `MOVE` command. The solution formulated by *Sleuth* generates a sequence of `MODIFY` commands. The first two perform useful work, but the third is redundant because *Sleuth* does not maintain full system state information. The `ENTER` command is a result of *Sleuth*'s random command selection. The `ENTER` command requires a sequencing rule: `ENTER` followed by one or more other commands followed by a `DRAIN`. Command #5 finally issues the `MOVE` command as required. The last two commands complete the test case. StorageTek testers interpret test cases like this from the point of view of testing a *shared device.* While this test case seems to have redundant or extraneous commands, the testers consider the sequence a merged list of instructions from multiple users.

The UCPOP test case must be generated in the context of the initial state and the goal. The initial state sets LSM 000 offline, Service level to Base, CAP 000 to Entering, and tape EVT280 located outside the ACS. The goal was to move a particular tape to LSM 001. The first two commands issued by the planner adjust the state of the system such that other commands are meaningful and can be executed. The `DRAIN` command is necessary because the initial state of the CAP is Entering. Since the CAP is a shared device, it must be released by the `DRAIN` command first. The `ENTER` command is important because tape EVT280 is currently located outside the robot tape library. After the tape is entered, the system issues the `MOVE` and achieves the goal.

An unexpected result of our comparisons was the kind of tests that were generated. The planner can potentially "discover" unusual command sequences to achieve the goal.

This is beneficial in test data generation because obvious approaches get tested most often and therefore find few or no faults. Unusual command sequences may achieve the same goal but uncover faults from command sequences that were not considered. For instance, one of our experiments required UCPOP to move a tape from one LSM to another. Instead of generating a `MOVE` command, it `EJECT`ed the tape from one LSM and `ENTER`ed it into the next. While this is a simple example, it shows how the planner can create innovative test sequences that the test engineers may not think about.

6.5 Conclusions

The *Sleuth* and AI Planner test generators for DBT show some of the possibilities for automated test generation. The hybrid implementation serves as a experimental test-bed for DBT test generation. New test generation algorithms, tools, or utilities can be inserted into *Sleuth* to experiment with the DBT test generation process. The AI planning system has been shown to be an effective approach to test case generation, too. It combines many of the semantic rules into the preconditions and effects of planner operators, and it can define test subdomains similar to *Sleuth.* Based on our comparison, there are tradeoffs between the planner and the hybrid test generation engines. *Sleuth* uses an efficient three staged test generation process. This is useful for incremental test generation, test case archive, and test case reuse. The *Sleuth* approach also requires much less specification and reliance on state information. When using *Sleuth*, testers focus on what commands are included in the test case. In contrast, the planner guarantees correct test cases based on state information, creates innovative and unusual test sequences, and focuses testers on the goals of test case generation leaving the planner to choose the appropriate commands. However, it requires a much more detailed specification and it does not tolerate ambiguous or incomplete information.

Chapter 7

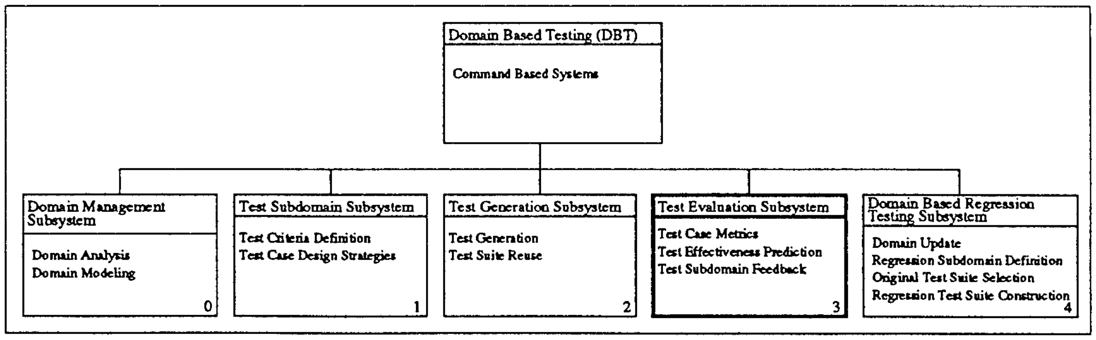
DBT EVALUATION

7.1 Introduction

This chapter presents the *Test Evaluation Subsystem* (see Subsystem 3 in Figure 7.1). We include a run-time performance evaluation of the *Sleuth* test generation tool and show how to evaluate the quality of test cases generated by DBT. The evaluation studied *Sleuth*'s run-time performance because test generation requires human assistance. Results show test generation time is reasonable for an interactive tool. We also show how a neural network predicts test case effectiveness. Attributes calculated from a test case and faults identified by the test case become the input and output vectors for a neural network, respectively. Once trained on these associations, the network predicts the effectiveness of new test cases.

7.2 Time Complexity Analysis of *Sleuth*

Sleuth is an interactive test generation tool developed at Colorado State University. *Sleuth* supports Domain Based Testing by providing tools and utilities to define domain



Domain Based Testing (DBT)
Command Based Systems
Domain Management Subsystem
Domain Analysis
Domain Modeling
0
Test Subdomain Subsystem
Test Criteria Definition
Test Case Design Strategies
1
Test Generation Subsystem
Test Generation
Test Suite Reuse
2
Test Evaluation Subsystem
Test Case Metrics
Test Effectiveness Prediction
Test Subdomain Feedback
3
Domain Based Regression Testing Subsystem
Domain Update
Regression Subdomain Definition
Original Test Suite Selection
Regression Test Suite Construction
4

Figure 7.1: Domain Based Testing Top Level Abstract Machine Diagram

models, create test subdomains, and to generate tests. It uses a hybrid representation for the domain model. The graphical user interface (GUI) was programmed using X Windows System X11R5 and Motif Release 1.2 [Bra92, O'r93]. The usefulness of a tool like *Sleuth* can be analyzed by inspecting the human-factors considerations such as the "look-and-feel" of the interface, the complexity of learning the tool, or its run-time performance. For this analysis, we examined the time complexity of test case generation. All experiments were run on a Hewlett-Packard 9000/735 Workstation with 128 megabytes of main memory and a 99 MHz clock. Tests were performed during normal, daily workstation use. Time is reported as the number of seconds between requesting a test case until the fully parameterized test case appears in the last window pane on the *Sleuth* main window. Our estimates show that the X Windows System and Motif overhead is no more than 10%.

Our experiment collected timing data from the StorageTek HSC Release 1.2 Domain, $D_0^{R1.2}$. Experiments used four *requested test case lengths* of 50, 100, 250, and 500 commands. We use the term *request test case length* because the *actual test case length* depends on the expansion of scripting rules. When all scripting rules are turned `Off`, the requested length equals the actual test case length. For each requested test case length, we measured test case generation time for three test subdomains. The first subdomain used the complete domain model, $D_0^{R1.2}$. The second test subdomain $TSD_{No-SR}^{R1.2}$ turned off all scripting rules, and the third subdomain $TSD_{NoRules}^{R1.2}$ turned off all semantic rules. More than likely, test engineers will seldom turn all the scripting rules off. Therefore, these experiments represent the upper and lower bounds for test generation time. All the timing data for these experiments can be found in Appendix B. We summarize the results in this chapter.

Consider test case generation from the complete domain model with requests for 500 commands. All commands can be generated, all scripting rules are turned on, and all intracommand and parameter constraint rules are turned on. In the ten generations of 500 commands, on average, *Sleuth* generated over twice as many commands as requested. This is explained by examining command sequencing rules. During the script expansion stage, a randomly generated command name is inspected for a script rule. If a script rule is defined and the rule is turned on, then it is invoked. In *Sleuth*, the script rule expands into a *sequence of commands.* This process may be recursive because one invocation of a script rule may generate another command that needs scripting rule expansion. Recursion is handled in *Sleuth* by setting a recursion limit. A limit of four was used for these experiments. This limit is not fixed and it can be set by the test engineer. Therefore, the variability in the actual length of the test cases depends on the number of script rules that are invoked and the recursion cutoff value.

Figure 7.2 shows the results from requesting 500 commands across all three test subdomains. The variability of the full domain model is apparent. When all scripting rules are turned off, there is no variability in the actual test case length. We analyzed this data using linear regression, and the regression line is drawn in the figure. The results of the regression show that *Sleuth* has a linear time complexity: $GenerationTime = 1.08 * Actual\ Test\ Case\ Length - 137.11$ with $r^2 = 0.985$. This suggests that *Sleuth* needs about one second per command to generate a test. The linear test generation time is a result of the recursion cutoff.

The results described for test case lengths of 500 commands were similar for experiments with 50, 100, and 250 commands. Figure 7.3 shows timing data for $D_0^{R1.2}$ across all four requested test case lengths. The regression shows a linear time complexity: $GenerationTime = 0.9 * Actual\ Test\ Case\ Length - 7.25$ with $r^2 = 0.984$. This result is consistent with the previous analysis. *Sleuth* takes about one second per command to generate a test case.

7.3 Using a Neural Network to Predict Effectiveness of Test Cases

One may wonder about the effectiveness of the DBT test generation method. Does it explore enough relevant faults to make its use worthwhile? Traditionally, a test suite is examined according to a *coverage* criteria. For Domain Based Testing, we could define *domain based coverage* measures such as: (1) Test all objects, (2) Test all parameters, (3) Test all script rules, or (4) Test all Intracommand Rules. Unfortunately, one

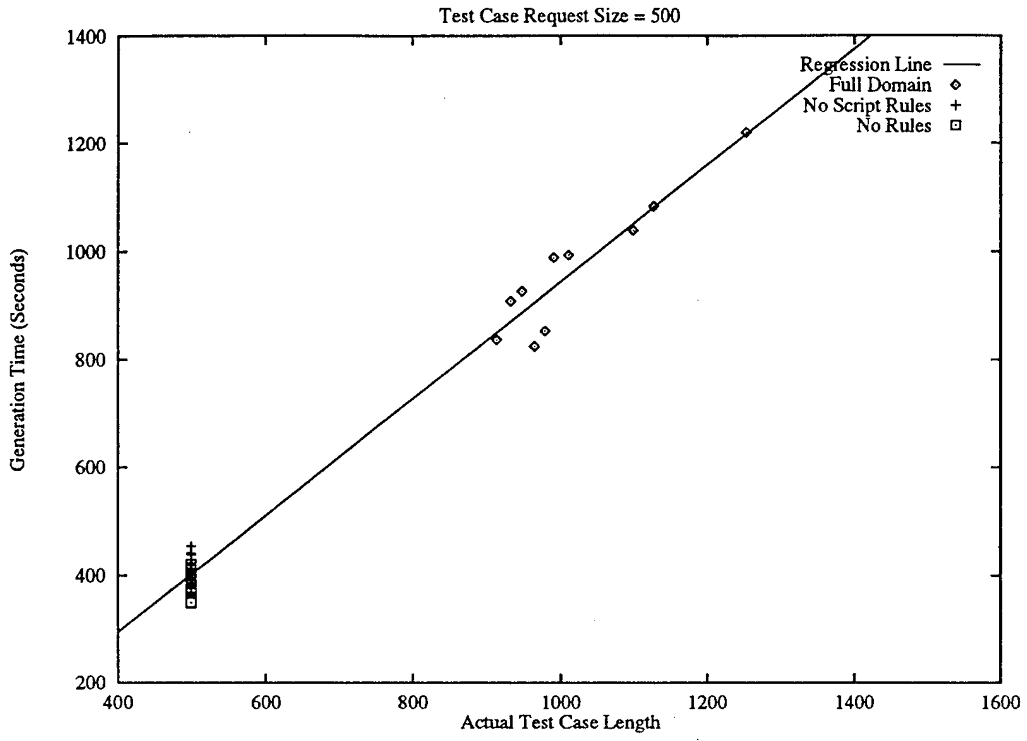


Figure 7.2: *Sleuth* Test Case Timing Study - Test Case Length = 500

coverage measure may not be sufficient to accurately measure test case effectiveness. For instance, effective tests for one application may require thorough object coverage while another application needs both object coverage and script rule coverage. In addition, software is in a continual state of change. Errors are fixed, new features or functions are added, and software fault profiles change as the software matures. As the software matures, the coverage measures that indicate effective test cases are likely to change. Therefore, we need a mechanism that (1) Evaluates test case effectiveness and (2) Adapts to new applications, software maturity, and software modifications. Figure 7.4 shows how to include test case effectiveness into the test generation process. Given a test case (T^{v}_{j-k}), a set of test case metrics and attributes are calculated, TM^{v}_{j-k}. Using TM^{v}_{j-k}, the *effectiveness predictor* predicts the fault exposure of the test case. Tests with low predicted effectiveness need not be run. This increases the fault yield (i.e., faults per test case).

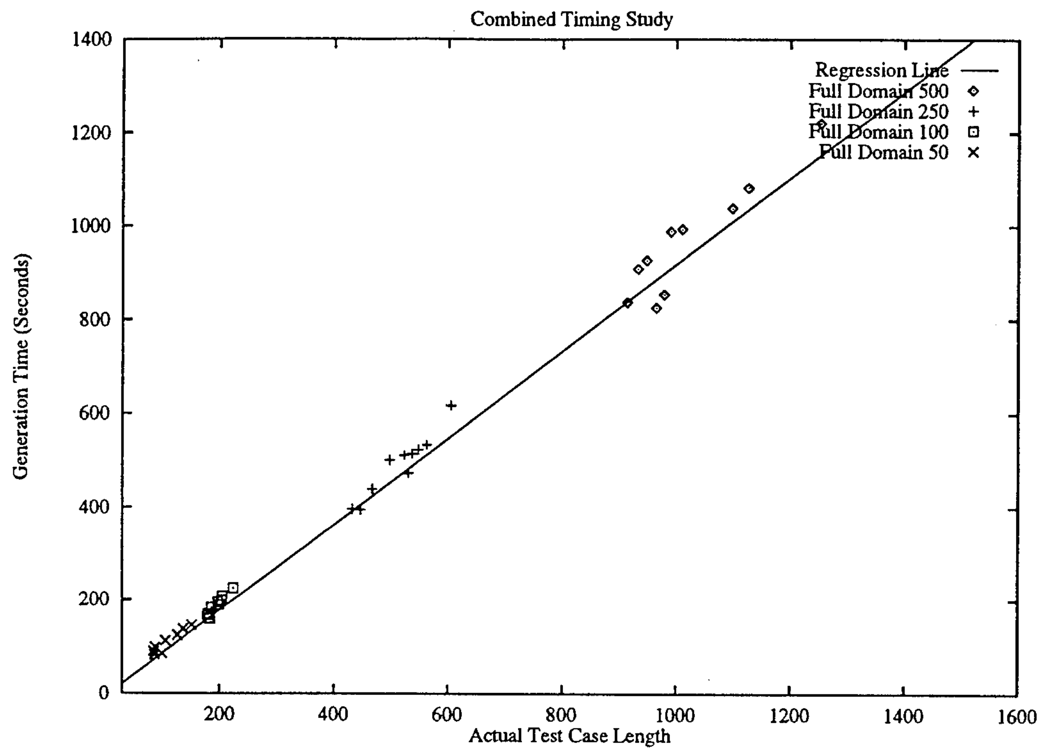


Figure 7.3: *Sleuth* Test Case Timing Study - Full Domain Test Data

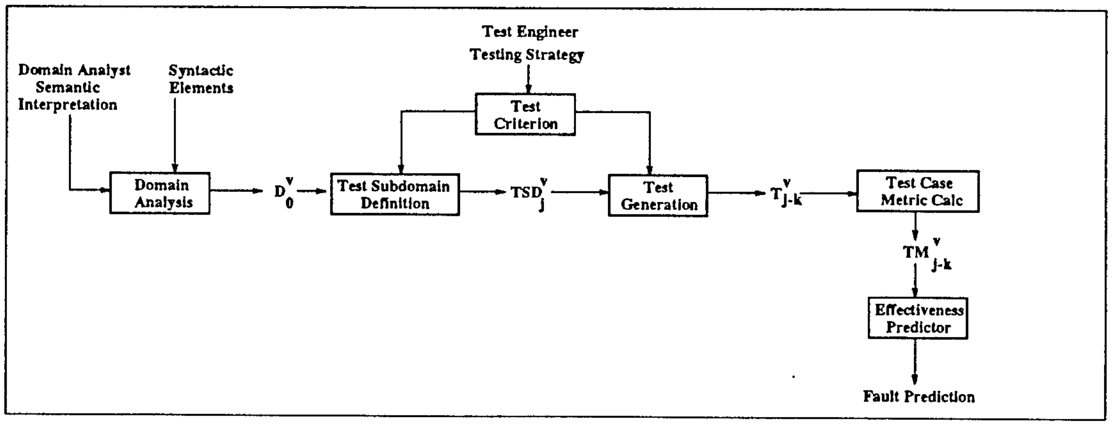


Figure 7.4: Test Generation Process Model with Effectiveness Predictor

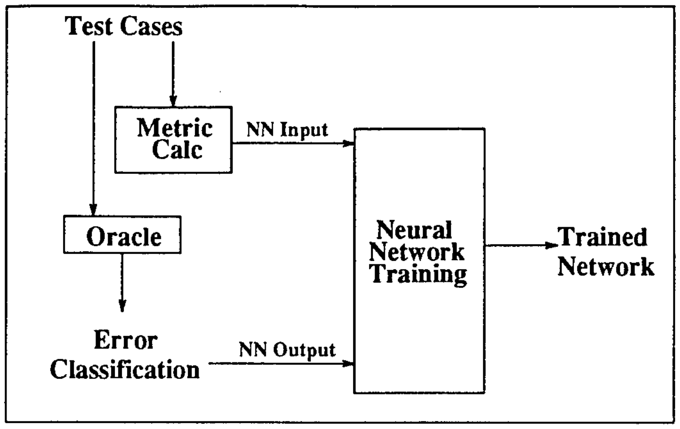


Figure 7.5: Neural Net Test Effectiveness Predictor : Training Phase

One solution to the test case effectiveness predictor is to use a neural network classifier [vMAM95]. We train the network to recognize relationships between test case attributes and the faults identified by a test oracle. Once trained, we no longer need the oracle. The network acts as a fault predictor when faced with a new test case. In the training phase, the network examines test case metrics and attributes as input and associates them with faults (neural network outputs). The output of the network is compared to the expected output (test oracle). An error is calculated and used to update the network weights. Training stops when the error is sufficiently low. After training, network weights are fixed and the net acts as a predictor. Given metrics and attributes from a new test case, the network predicts its fault exposure.

If the test case does not expose faults, then we need not run it. Figure 7.5 shows how to train the neural network. Test case metrics are extracted from the test case. Metrics measure test case length, command frequencies, and parameter use frequencies. An "oracle" classifies any errors exposed by the test case. The oracle is an objective arbiter to whether a given test case exposes a fault. In practice, testers act as arbiters, but in our controlled laboratory experiment we defined a synthetic test oracle.

Test case metrics and the error classification are used to train the neural network. Once trained, we use it to predict the fault exposure of new test cases. Figure 7.6 shows how we evaluate the effectiveness of the neural classifier. Given a test case, the

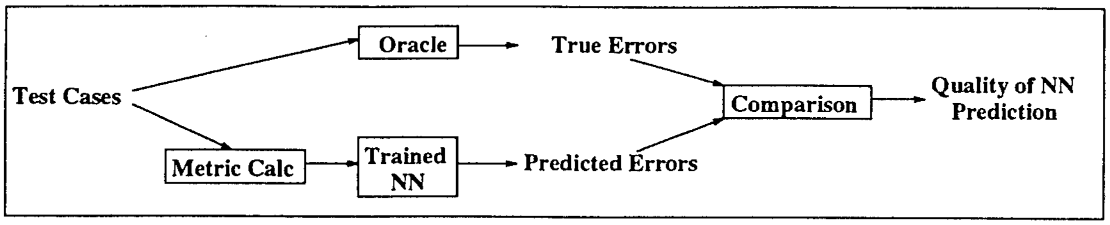


Figure 7.6: Neural Net Test Effectiveness Predictor : Evaluation Phase

test oracle identifies the actual faults exposed by the test case and the neural network predicts the fault exposure. Comparing the two, we measure how well the neural net acts as a test case effectiveness predictor.

7.4 Experiment Design

We conducted an empirical study to show the effectiveness of a neural net fault predictor. The study used the DBT test generation tool *Sleuth* to generate test data. Using test case metrics, a synthetic test oracle evaluated each test case for error classification. The neural net trained on test metric input patterns and mapped them to the test oracle's error classification. Once trained, the network acts as a test case effectiveness predictor.

7.4.1 Test Data Generation

The empirical study needed a data set to train and evaluate the neural net. Using the *Sleuth* test generator, we defined six test subdomains from which 180 data vectors were generated. Each test subdomain is associated with a set of test criteria. For the experiment, we trained and tested the neural network on a subset of commands from the StorageTek robot tape library. Table 7.1 lists six subdomains used to generate training and test data. We also show test oracles the test case is likely to invoke (see next section for test oracle descriptions). The first subdomain, D_0^{NN}, is a command subset from the HSC command language. All semantic rules are turned on. The second test subdomain TSD_{NoMD}^{NN} turns off the `Mount-Dismount` scripting rules. The robot

Table 7.1: Test Subdomains Used in Neural Network Prediction Experiments

	Test Subdomain	Possible Faults
1	StorageTek Subset (D_0^{NN})	Oracles 1,4-7,10
2	No Mount-Dismount Rules (TSD_{NoMD}^{NN})	Oracles 2,4-7,10
3	No Enter-Drain Rules (TSD_{NoED}^{NN})	Oracles 1,3,4-7,10
4	No Script Rules (TSD_{NoSR}^{NN})	Oracles 2-7,10
5	No Intracommand Rules (TSD_{NoICR}^{NN})	Oracles 1,4-10
6	No Semantics Rules ($TSD_{NoRules}^{NN}$)	Oracles 2-10

tape library requires `Mount` and `Dismount` commands to be sequenced properly. It does not make sense to issue a `Dismount` request unless a tape was `Mounted` earlier in the command sequence. Test subdomain TSD_{NoED}^{NN} tests for faults in `Enter-Drain` sequences. The `Enter` command allows a test engineer to insert new tapes into the robot tape library through a special door called a Cartridge Access Port (CAP). The door is a shared resource and it is assigned to one tester at a time. When finished, the tester issues a `Drain` command to release the door for the next tester. In test subdomain four, TSD_{NoSR}^{NN}, all scripting rules were turned off. By turning off all scripting rules, the test generator creates random lists of commands, but it still follows all parameter value selection rules. TSD_{NoICR}^{NN} turns off intracommand rules. Intracommand rules specify how to choose parameter values within a single command. Sometimes the value of one parameter constrains the choices of other parameters in the same command. For instance, in the StorageTek robot tape library, when moving tapes within the same "silo" the source panel number must be different from the destination panel number. The last test subdomain, $TSD_{NoRules}^{NN}$, turns off all semantic rules. This effectively generates random sequences of commands and parameter values. Thus, the six test subdomains represent a variety of sensible and pathological system uses.

7.4.2 Test Oracle

The test oracle acts as an impartial, objective arbiter to determine whether a given test case (i.e., sequence of commands) exposes a fault and if so, the type of fault. The oracle for this study was synthetic in that it did not model actual software behavior [1]. It provides a controlled laboratory environment to analyze the neural network results. When used in the field, the test oracle is replaced by testers who judge the results of running a test case.

Table 7.2 specifies characteristics of ten hypothetical faults in the HSC, representing six different aspects of the application domain. These ten hypothetical faults cause failures at three different severity levels. Severity 1 is the most severe and Severity 3 the least. Severity 4 represents no-fault. For each fault, Table 7.2 gives the symptom and the specification how to recognize it (fault indicator column), as well as the severity of the problem. The first three faults focus on improper command sequences. Oracle #1 observes correctly ordered `Mount-Dismount` commands but it indicates a low severity fault because of a slight error in the mount scheduler. The second oracle identifies a condition where there are more `Dismount` requests than `Mount` requests. The third sequencing problem identifies the condition when a user does not release the Cartridge Access Port correctly.

The chronic command fault identifies problems in the Control Data Set (CDS). The CDS maintains a database about tape locations in the ACS. Here, long test sequences and too many `Disable` requests cause database inconsistencies. The next two faults focus on a chronic command fault coupled with a particular parameter value. If the `MODIFY` command is issued to turn LSM 000 `Online`, it will not work properly. If the frequency of the request is less than three, then it takes the system an unusually long time to change the LSM's state (Severity 3). If more than three requests are issued, then the LSM stays `Offline` and the fault is more severe (Severity 2). The seventh fault focuses on the Pass Through Port (PTP) between two LSM's. If we have a long test case and there are more than eight PTP `Move` requests, then a tape can be lost. Faults eight and nine relate to intracommand rules. For a short test case, the system fails to

[1]It does not represent actual faults in the HSC product.

Table 7.2: Test Oracle Specification

Test Oracle Specification				
Domain Model Focus		Symptom	Fault Indicators	Sev
Sequencing Fault	1	Incorrect Mount Order	Test Case Length $\geq$ 30 MOUNT Freq = DISMOUNT Freq	S3
	2	Incorrect Command Sequencing	Test Case Length $\geq$ 20 MOUNT Freq < DISMOUNT Freq	S2
	3	CAP isn't released	Test Case Length $\geq$ 10 ENTER Freq > DRAIN Freq	S2
Chronic Command Fault	4	Inconsistent Database	Test Case Length > 30 CDS Freq $\geq$ 5 `disable` Parameter > 3	S1
Chronic Command & Parameter Fault	5	Unusually High Delay to put 000 Online	`MODIFY 000 ONLINE` MODIFY Frequency $\leq$ 3	S3
	6	System Stays Offline	`MODIFY 000 ONLINE` MODIFY Frequency > 3	S2
Object Fault (Pass Through Port)	7	Lost Tape	Test Case Length > 30 Number of PTP Moves > 8	S1
Intracommand Rule Fault	8	No warning issued for violating the rule.	Test Case Length $\leq$ 15 LSM1 = LSM2 AND Panel1 = Panel2	S3
	9	Tape is moved to wrong destination panel	Test Case Length > 15 LSM1 = LSM2 AND Panel1 = Panel2	S2
Command Interaction	10	Inconsistent Database	Ratio: MOUNT Freq to MOVE Freq > 0.8 Test Case Length $\geq$ 30	S1

issue a warning message for violating the rule. If the test is long, the tape is moved to the wrong destination panel. The last fault occurs with interacting commands. If the ratio of `Mount` and `Move` commands is greater than 0.8, the Control Data Set (CDS) becomes inconsistent.

7.4.3 Neural Network Training

We trained four neural nets using error backpropagation with unipolar sigmoid units. Each network learned to predict one fault severity level. We used 21 input nodes and one output node for each network. The number of hidden units was calculated experimentally to achieve the best Root Mean Square (RMS) Error during training. Network training started with random initial weights between -0.005 and +0.005. Before training, all input and output vectors in the data set were normalized using a linear scale [2]. The minimum value for each input parameter is set to 0.0 and the maximum value is set to 1.0. Desired output values (called targets) are set to 0.1 for a "zero" response and 0.9 for a "one" response.

The test data set for neural net training included 180 observations, thirty test cases from each of the six test subdomains. Table 7.3 shows an example test case generated from TSD^{NN}_{NoED}. Table 7.4 shows the 21 metrics used as input to the neural net and lists the four output severity levels. The first input calculates the length of the test case, the next ten inputs represent the frequency of each command in the test case, and the last ten inputs count the number of unique values for each parameter. The output vector identifies fault severity levels 1-4. Table 7.4 also shows a data vector for the example test case in Figure 7.3. As indicated, two faults are exposed by this test case. The first identifies a Severity 2 fault because the `ENTER` command is not followed by a `DRAIN` command. The second fault identifies problems with the "`MODIFY 000 Online`" command.

We trained all four networks (one for each severity level) using the Leave-One-Out-Method (LOOM). LOOM removes one vector from the data set. The single vector is called the *test vector* and the remaining patterns (e.g., 179 vectors for our experiments) are called the *training set.* We train the network using the training set and evaluate its classification on the test vector. The choose-one-test-vector, train, evaluation cycle is performed for each vector in the data set, 180 in all.

LOOM is useful for training any neural network, but the amount of computation is prohibitive for large data sets. An alternative training method is called data-splitting which divides the data set into a training set and a test set. Data splitting typically requires a large data set for training. For instance, one rule-of-thumb uses the following equation to determine the size of the training set [Fau94]:

$$P = \frac{W}{e} \tag{7.1}$$

[2]Scaling reduces the side effects of scale differences between parameters. Linear, square root, logarithm, and general data transformation are typical scaling methods [ST80].

Table 7.3: Test Case Generated from TSD_{NoED}^{NN} for the Neural Network Experiments

Test Case
MOVE (EVT393) Tlsm (000)
DISPLAY VOLSER EVT199
CDS Disable STandby
MODIFY 000 OFFline
ENTER 001
DISPLAY CAP
MODIFY 001 ONline
MODIFY 000 ONline
DISPLAY LSM 000
MOUNT EVT180 A17
EJECT EVT180 001
EJECT EVT185 010
MOVE (EVT199) Tlsm (000)
DISPLAY CAP
DISMOUNT EVT180 A17
SRVLEV BASE
DISPLAY SRVLEV

Table 7.4: Input Vector and Output Vector Description

Vector Index	Description	Example Data Vector
Input		
1.	Test Case Length	17
2.	CDS Frequency	1
3.	DISMOUNT Frequency	1
4.	DISPLAY Frequency	5
5.	DRAIN Frequency	0
6.	EJECT Frequency	2
7.	ENTER Frequency	1
8.	MODIFY Frequency	3
9.	MOUNT Frequency	1
10.	MOVE Frequency	2
11.	SRVLEV Frequency	1
12.	acs Frequency	0
13.	cap Frequency	2
14.	cc Frequency	0
15.	drive Frequency	1
16.	dsn Frequency	0
17.	host Frequency	0
18.	lsm Frequency	2
19.	pp Frequency	0
20.	rr Frequency	0
21.	volser Frequency	4
Output		
22.	Severity 1 Indicator	0
23.	Severity 2 Indicator	1 (oracle 3)
24.	Severity 3 Indicator	1 (oracle 5)
25.	Severity 4 Indicator	0

Table 7.5: Network Topologies

Severity	Input Units	Hidden Units	Output Units
S1	21	10	1
S2	21	2	1
S3	21	1	1
S4	21	1	1

where P is the size of the training set, W is the number of weights to train, and e is the accuracy of the classification. Consider a network designed for the test case effectiveness prediction with 21 input units, 2 hidden units, and a single output unit. Using the equation above, we calculate that the training set must contain at least 440 patterns to be assured of classifying 90% of the test patterns.

$$P = \frac{44}{0.1} = 440 \tag{7.2}$$

For small data sets, the data-splitting technique may not leave enough data in the training set to be useful. So, LOOM is most useful for small data sets. We used LOOM to experimentally calculate the best network topology (number of hidden units). Table 7.5 lists the best topology for each network. Fault Severity 1 needed ten hidden units, Severity 2 needed two, and Severity 3 and 4 used one hidden unit.

7.5 Evaluation

The results from training all four networks are listed in Table 7.6. These data should be interpreted as a conservative estimate of a neural network's predictive abilities because of the small data set. The second column shows how well each network predicted individual fault severities. The Severity 1 (most severe fault) predicted the best with 95% and the second best predictors were Severity 2 and Severity 4 networks with 83%. We placed the incorrectly classified tests into one of three categories: *False Positive*, *False Negative*, and *Remaining Misclassification.* A False Positive response is recorded when the network predicts a fault that doesn't truly exist. A False Negative response

Table 7.6: NN Prediction Results

Severity	Correctly Classified	False Positive	False Negative	Remaining Misclassification
S1	172 (95.6%)	0	0	8
S2	150 (83.3%)	2	2	26
S3	132 (73.3%)	6	9	33
S4	151 (83.8%)	0	0	29

is recorded when the neural net predicted Severity 4 (no fault exposed) when the test oracle indicates a fault. Remaining Misclassification refers to tests that were classified by the neural net as exposing a fault, but of the wrong type. We use this information to analyze three test data generation objectives.

7.5.1 Test Data Generation Objective 1: Reduce Number of Test Cases

One goal for test data generation is to reduce the number of test cases run on the system under test. Each test case consumes machine time and resources. In addition, testers must evaluate each test case to determine whether a fault was exposed. To reduce testing time and cost, we need to run tests that are likely to identify a fault. This is particularly important with automated test data generation systems that easily and quickly generate thousands of test cases.

A tester could use the neural net classifier to reduce the number of test cases. Severity 4 predictions need not be run. Two things that help us achieve this objective are automated test data generation and low cost of test case evaluation. Automated test data generators like *Sleuth* generate test cases quickly. They can generate tests much more quickly than it takes to run them. Likewise, a neural network predictor evaluates a test case in a single forward-pass through the net. The key to meeting this objective is a predictor that is good at the decision: `Does this test case expose a fault or not?` We aren't concerned about misclassification of the individual fault severity but we must keep *False Positive* predictions to a minimum.

Our empirical study with the StorageTek tape library shows that the neural net has a low False Positive prediction rate. The network identified eight tests out of 180 that should expose faults while in actuality they would not. This suggests that the neural net can be used to reduce the number of test cases.

7.5.2 Test Data Generation Objective 2: Emphasize Severe Error Exposure

Another goal for test data generation is to create tests that expose the most severe system faults. This is important because the most severe faults cause system failure or reduced system operations. The cost to fix such problems once the product is in the field is high. To make matters worse, severe faults are often the most difficult to isolate, identify, and expose.

A tester could meet this test data generation objective using a neural net to evaluate tests before they are run. If the net predicts high severity (e.g., Severity 1 or 2 for our tests), then the test should be run. The key to achieving this objective is an accurate fault prediction at the higher levels of severity. We are concerned about misclassification of tests at lower severity levels. If the net tends to classify test cases with a lower fault severity than truly exists, then tests will not run that should. If misclassified with higher fault severity than truly exists, then we will run tests that need not run. We also require the neural net to have a low *False Negative* prediction. A False Negative prediction incorrectly classifies tests that are likely to expose faults as Severity 4. We will not run them, but we should.

Data from the empirical study suggests that the neural network can be used to identify tests for severe error exposure. The network predicts Severity 1 very well. Its Severity 2 classification not as good, but the False Negative rate is low. Only eleven out of 180 vectors were False Negative. Studying the Remaining Misclassified patterns, we identified four tests where the network predicted a higher severity fault than really existed.

7.5.3 Test Data Generation Objective 3: Minimize Number of Test Cases

Another test data generation goal is to minimize the total number of test cases to run. This can be viewed as a combination of the first two objectives. First, we need to reduce the number of tests run and we also need to accurately predict the fault exposure at each severity level. We keep test cases that expose errors at multiple severity levels.

One way to rank a test case as more effective than another is to count the number of fault severity levels the test exposes. If one test exposes multiple severity levels (say Severity 2 and 3), it should be ranked higher than a test that exposes a single level (i.e., Severity 2). A test case that exposes multiple severity levels meets several test criteria, it reduces the number of test cases to run, and it *minimizes* the total test set for the system under test. To achieve this objective, we need accurate test case effectiveness prediction. We emphasize the accuracy of the neural net prediction because we want to keep those tests that identify multiple severity levels. If the net misclassified a multi-severity test as a single severity test, then it may not be ranked high enough to be included in the test set.

The data set for the empirical study contains 59 multi-severity vectors. The neural net fails to predict every fault in 20 multi-severity test vectors. When failing to predict every fault, these 20 vectors could be ranked incorrectly and some of them could be eliminated from a test run. This data shows that the neural net is better at "coarse" classification (i.e., Test Data Objectives 1 and 2) than for detailed test effectiveness prediction. The test case metrics used by the neural net is one source of variability in the effectiveness predictions. This is not surprising, since the test oracle has much more precise information through the error specification of Table 7.2 than the neural net which only gets summarized information via the metrics of Table 7.4. What these results indicate, however, is that the neural network produces remarkably good predictions despite limited metric information. In particular, the network effectively screens no-yield test cases and identifies high severity tests.

7.6 Summary

Based on our experiments, the *Sleuth* hybrid test generation engine is an efficient test case generator. The time to generate a test case is reasonable for an interactive tool (e.g., about one second per command generated). We also show how a neural network is a approach to test case effectiveness prediction. The neural net formalizes and objectively evaluates some of the testing folklore and rules-of-thumb that are system specific and often require many years of testing experience. A neural network is neither system nor test case metric specific. Therefore, it can be used with a variety of test generation methods, test case metrics, and fault severity levels.

Chapter 8

DOMAIN BASED REGRESSION TESTING

8.1 Introduction

The operation and maintenance phase of the software life cycle addresses changes and modifications to a software product. Modifications correct errors, add new features and functionality, or improve software performance or resource use. As software changes, we retest it to make sure the modifications work and original features are not broken. Test engineers are likely to exploit the original set of test cases, but they often cannot rerun *all* of them. Rerunning all tests may take too much time, some of them may no longer apply, and new cases may be needed to test modifications or new features. Testing software changes is called *regression testing*, and the set of test cases run during a regression test is called the *regression test suite*. Domain Based Regression Testing (DBRT) is the process and method of creating regression test suites based on information from a domain model. This chapter describes Domain Based Regression Testing and relates it to the Test Generation Process Model (see Subsystem 4 in Figure 8.1).

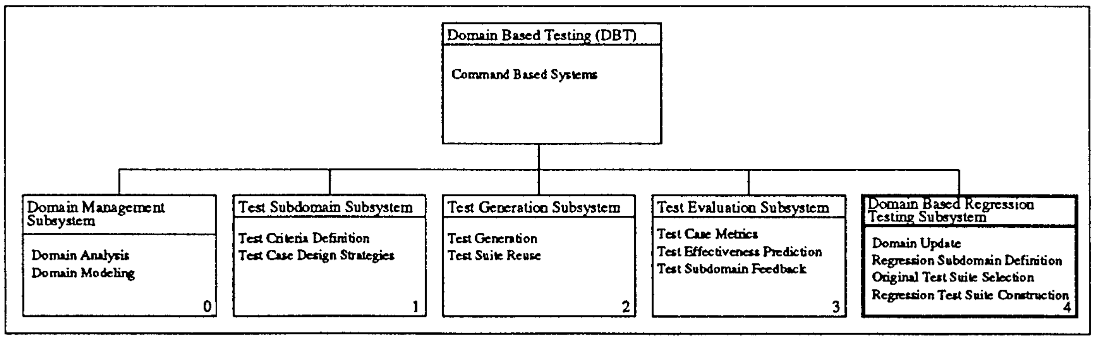

Domain Based Testing (DBT)
Command Based Systems
Domain Management Subsystem
Domain Analysis
Domain Modeling
0
Test Subdomain Subsystem
Test Criteria Definition
Test Case Design Strategies
1
Test Generation Subsystem
Test Generation
Test Suite Reuse
2
Test Evaluation Subsystem
Test Case Metrics
Test Effectiveness Prediction
Test Subdomain Feedback
3
Domain Based Regression Testing Subsystem
Domain Update
Regression Subdomain Definition
Original Test Suite Selection
Regression Test Suite Construction
4

Figure 8.1: Domain Based Testing Top Level Abstract Machine Diagram

8.2 Overview of Domain Based Regression Testing

The DBT Test Generation Process provides test case reuse opportunities. Chapter 5 developed three reuse scenarios and described their use in an industrial setting. These scenarios provided mechanisms to exploit tests cases for a particular version of the system under test. After closer inspection, we wanted to refine test suite reuse strategies when the system under test is modified. These refinements are captured in a regression testing process based on a domain model. For this research, regression testing rules are associated with types of command language modification. For each type of modification, a four step regression test process is defined: *Regression Domain Definition*, *Regression Subdomain Definition*, *Test Suite Selection*, and *Regression Test Suite Construction.*

Regression Domain Definition modifies the original domain model based on changes to command language syntax and semantics. The result is a *regression domain* with changes to the script, command, and parameter definitions. The second step defines *regression subdomains.* Regression subdomains are similar to test subdomains in that they specify a test criteria for test case generation. They are more specific than test subdomains because they focus test generation on command language modifications. The third step, Test Suite Selection, updates original tests so they conform to the new domain. Original tests can be test scripts, test templates, or test cases. A test script is a list of command names. A test template is a list of commands with place holders for parameters. A test case is a list of fully parameterized commands. The fourth step, Regression Test Suite Construction, defines rules for combining original tests and new tests into a regression test suite. Regression test suite construction can be *minimal* or *maximal.* A minimal rule relates to a less stringent regression test criteria where we assume the impact of the software modification is small. Thus, the size of the regression test suite tends to be small. A minimal approach tries to reduce the need for regression testing and is appropriate for systems with high existing reliability and for systems where the effect of the changes is minor. (e.g., as assessed by code impact metrics like [YC80]). Maximal regression test strategies "assume the worst" with respect to the impact of the software modification. This is a stronger regression test criteria that leads to more extensive regression test suites. Regression tests consider more aspects of domain model changes; hence, the size of a maximal regression test suite tends to be large.

8.3 Integrating Regression Testing with Test Generation

Figure 8.2 shows the IPO diagram for DBRT. This diagram extends the Test Generation Process Model in Figure 5.3 in Chapter 5. Table 8.1 defines each set in the figure. The first DBRT step, *Domain Update Rules*, transform the original domain model, D_0^v, into a *regression domain*, RD_i^v. The regression domain captures syntax and semantic modifications to the original command language for regression testing purposes. Superscript v denotes the version of the system under test and the subscript denotes change i to the command language. Consider adding a new REPORT command to the StorageTek HSC domain, $D_0^{R1.2}$. Regression domain $RD_{report}^{R1.2}$ includes the HSC domain model and it adds the syntax and semantics of the REPORT command. This may include new command sequencing rules, intracommand rules, and parameter constraints.

The second transformation required for DBRT defines *Regression Subdomain(s)*, RSD_{i-l}^v. Regression subdomains configure the regression domain to focus test generation on the command language change. We build RSD_{i-l}^v from RD_i^v by mechanisms like restricting the members of scripting classes, turning semantic rules `on/off`, or adjusting command generation frequencies. Different regression test criteria define different regression subdomains; hence, the need for subscripts $(i - l)$. By convention, we use the subscript number 1 (i.e., $i - 1$) to denote the first regression subdomain. The first regression subdomain should include all test criteria to test the command language modification. Other regression subdomains can be defined to test other aspects of the command language changes. For instance, $RSD_{report-1}^{R1.2}$ and $RSD_{report-BV}^{R1.2}$ denote two regression subdomains for the new REPORT command. The first subdomain configures the regression domain to test the new command, its semantic rules, and parameter values. $RSD_{report-BV}^{R1.2}$ is a specialized regression subdomain that specifies boundary-value criteria for REPORT parameters.

The third DBRT step examines original test suites, T^v, as potential sources for regression test cases. More than likely, we will not be able to use some of the original tests, some may need modification before we can use them, and a few can be reused without change. Leung and White show how to address this problem by partitioning the set of original test cases into subsets [LW89]. We cannot use their partitions directly, but we can leverage the concepts for DBRT. The lower half of Table 8.1 lists the four test case partitions for DBRT: $Discard_i^v$, $Reuse_i^v$, Mod_i^v, and $Regen_{i-l}^v$.

$Discard_i^v$	Tests not applicable to regression domain i.
$Reuse_i^v$	Tests usable immediately in regression domain i.
Mod_i^v	Tests requiring minor modifications. Consider a command deleted from the command language. Original tests that contain the deleted command can be used in regression tests by removing the obsolete command.
$Regen_{i-l}^v$	Test scripts and test templates applicable to regression domain i, regenerated using regression subdomain RSD_{i-l}^v. Test scripts and templates need regeneration because they are not fully parameterized test cases.

Three points must be considered: One, the tests in the *Reuse*, *Modify* and *Regeneration* sets are *sources* for regression test suites. They do not have to be used. Two, the test suite selection rules are *dependent on the regression domain* RD_i^v. The subscript i in each set description corresponds to the regression domain. Therefore, the four subsets must be recalculated for each regression domain, RD_i^v. Three, modification of tests in Mod_i^v and $Regen_{i-l}^v$ are not automatically modified or regenerated. They are processed only if selected for regression testing.

The last stage in DBRT constructs regression test suites by combining tests from the *Reuse*, *Modify*, and *Regeneration* sets. Testers also include new test cases, Tr_{i-l-m}^v $(m = 1,2,3,\ldots)$, for regression testing. New tests use information from the regression subdomain (RSD_{i-l}^v) and test criteria guidance from the test engineer. Depending on how the tester chooses tests from the four sources, a variety of regression test suites can be constructed ranging from *minimal* tests to stricter *maximal* tests.

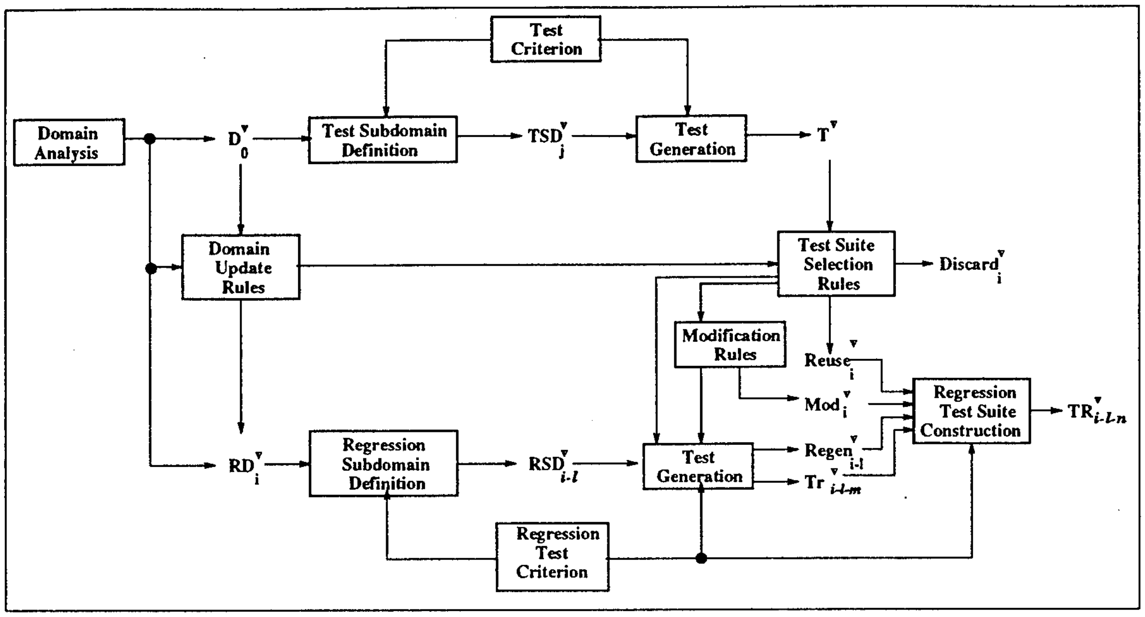


Figure 8.2: Domain Based Regression Testing - IPO Diagram

Table 8.1: DBRT Definitions

Set	Definition
D$_v^0$	Domain Model for version *v*
TSD$_j^v$	Test Subdomain *j* for version *v*
T^v	Test Suites for version *v*
RD$_i^v$	Regression Domain Model for version *v* and change *i*
RSD$_{i-l}^v$	Regression Subdomain *l* for RD$_i^v$
$Discard_i^v$	Tests no longer applicable to the Regression Domain
$Reuse_i^v$	Original Test Suites that are used with no changes
Mod_i^v	Tests that need modification.
$Regen_{i-l}^v$	Tests that need regeneration.
Tr$_{i-l-m}^v$	New Test Suite *m (m=1,2,3,...)* from RSD$_{i-l}^v$
TR$_{i-l-n}^v$	Regression Test Suite *n (n=1,2,3,...)* for RSD$_{i-l}^v$

8.3.1 Command Language Modifications

This research is based on a domain model representation of a command language user interface. We use the domain model to automatically generate tests. Regression testing relates changes, modifications, and updates in the command language UIF to changes, modifications, and updates to the domain model. The regression process uses the updated domain model to generate regression test suites. We classify command language changes into four categories, (1) Delete an old or obsolete command, (2) Add a new command, (3) Modify a command - Delete part of a command, and (4) Modify a command - Add new parts to a command. For clarity, we use the notation, RD_i^v *where (i=del,add,mod_del, mod_add)* for each of the four language modifications.

Each command language change requires separate rules for developing regression test suites. In the following sections, we explain regression test generation rules for each command language modification. We emphasize that each of the four modifications can occur more than once in a command language upgrade. The rules presented below describe regression testing procedures for a single instance of the modification and should be aggregated for sets of changes.

8.4 Deleting a Command

8.4.1 Regression Domain Definition

Sometimes old or obsolete commands are removed from a command language. Old commands remain in interface languages to support "legacy" systems or previous releases of the software. Over time, these commands are phased out. Deleting a command from a command language and removing associated code in a software system can cause problems if the modification is not tested properly. When a command is deleted from the command language, all syntactic and semantic information must be removed from the domain model description. The list below shows the necessary actions to update a domain model when a command is deleted from the command language. The result is regression subdomain RD_{del}^v.

Script Class	Command name is removed from all script classes.
Script Rule	Delete scripting rules named for the deleted command. Remove command name from all remaining script rules.
Command Syntax	Remove syntax definition.
Intracommand Rules	Remove intracommand rules defined for the deleted command.
Pre/Post Conditions	Remove all pre/post conditions for the deleted command.
Parameters	Parameters (parameter value sets) unique to the command are removed. If a deleted parameter is part of a parameter constraint rule, remove the parameter constraint.

8.4.2 Regression Subdomain Definition

The second step in the regression test process is to define *regression subdomains.* Steps to configure the first subdomain, RSD^{v}_{del-1}, are listed below. The test intent is to focus testing on the parts of the domain model influenced by the deleted command.

- Define Set P such that P contains all parameters of the obsolete command that are not unique to this command. Set P thus provides an indicator about the degree of interaction between the obsolete command and existing objects.
- Define Set C such that C contains all commands that use parameters in P. Set C defines all ways that existing commands use objects in common with the obsolete command.
- Define Set SC such that SC contains all script class names that contain commands in C.
- Define Set SR such that SR contains all script rules that use commands in C.

All scripting classes in SC are used and all scripting rules in SR are turned on. At the command level, all intracommand rules for commands in C are turned on. All parameter constraints defined in the parameter value files are turned on.

8.4.3 Test Suite Selection

The general approach for test suite selection is to remove all occurrences of the deleted command from the test scripts, test templates, and test cases. Table 8.2 relates each test archive with a test suite partition and a modification rule. A test case is placed in the *Reuse* set if it does not contain the obsolete command. These can be used immediately in a regression test suite. A test case that contains the deleted

Table 8.2: Delete a Command : Test Suite Selection and Modification Rules

Archive Type	Reuse	Mod	Regen
Test Script	$\emptyset$	$\emptyset$	Remove name of deleted command
Test Template	$\emptyset$	$\emptyset$	Remove template of deleted command
Test Case	Tests that do not contain the deleted command	Remove all occurrences of the command from the test case.	$\emptyset$

command is placed in the *Modify* set. The modification rule removes all occurrences of the deleted command from the test case. Test scripts and test templates are placed in the *Regeneration* set because they are not complete test cases. If a test script or test template contains the deleted command, then all occurrences of the command are removed. Regeneration recalls the test script or test template and processes it through the test generator. All test scripts and test templates use current information from the regression subdomain. Scripts and templates are not automatically regenerated. Regeneration takes place when a test script or test template is requested during regression test suite construction.

8.4.4 Regression Test Suite Construction

Regression test suite construction combines new tests and tests from the *Reuse*, *Modify*, and *Regeneration* sets to create regression test suites. The way in which tests from these sets are combined determines how aggressive or conservative the regression test is. Table 8.3 shows three levels of regression test suites ranging from a minimal test to a maximal test. This table represents regression tests constructed from RSD^{v}_{del-1}. Tables similar to this one can be used for other regression subdomains. A minimal regression test suite chooses tests solely from the *Reuse* and *Modify* sets. This provides an immediate set of regression tests for the updated command language. If there are few tests in either set or if a more thorough regression test is required, a more conservative regression test includes tests regenerated through the test generation process. Finally, a maximal regression test may be needed for high reliability or to test a high impact

Table 8.3: Delete a Command - Regression Test Suite Construction Schemes

Regression Test Suite Source	RSD^v_{del-1} (Minimal)	RSD^v_{del-1}	RSD^v_{del-1} (Maximal)
Reuse	X	X	X
Modify	X	X	X
Regeneration		X	X
New Test Cases			X

change to the command language. The maximal regression test suite chooses tests from the reuse, modify, and regeneration sets and it adds *new test cases* generated from the regression subdomain.

8.5 Adding a New Command

8.5.1 Regression Domain Definition

As a software system matures, new features or functions are added. These functions may require new commands. New test cases specifically designed for the new command are needed, and the original *test scripts* may be used as a source for regression testing. Depending on the command and its relationship to existing objects in the domain model, adding a command to the command language can result in considerable modifications to the syntax and semantics of the domain model. When adding a new command, syntax and semantic information related to the command is entered into the regression domain. The list below shows the necessary actions to update a domain model when a command is added to the command language. The result is regression domain RD^v_{add}.

Script Class	Add new command name to appropriate script classes. Create new script classes if necessary.
Script Rule	Add new script rule for the new command if necessary. If needed, update old script rules with new command.
Command Syntax	Enter new command syntax.
Intracommand Rules	Add intracommand rules as required.
Pre/Post Conditions	Include pre/post conditions for new command.
Parameters	If new command introduces new parameters, Create new parameter value sets. Add parameter constraint rules as needed.

8.5.2 Regression Subdomain Definition

In the second step of the regression test process, regression subdomain(s) are defined. For this command language modification, we will demonstrate two different test subdomains. This illustrates how different regression test criteria can be added to DBRT. The first regression subdomain, RSD^{v}_{add-1}, defines the minimal criteria to test the new command.

- Let NC be the name of the new command.
- Define Set SC such that SC contains the names of all scripting classes that contain NC.
- Define Set SR such that SR contains the names of all scripting rules that contain NC.

The second regression subdomain, RSD^{v}_{add-2}, is a more strict subdomain that includes all changes to the domain model.

- Define NC to be the new command.
- Define Set P such that P contains all parameters of NC. Set P provides an indicator about the degree of interaction between the new command and existing commands.
- Define Set C such that C contains all commands that use parameters in P. Set C defines all ways that existing commands use objects in common with the new command.
- Define Set SC such that SC contains all script class names that contain commands in C.
- Define Set SR such that SR contains all script rules that use commands in C.

8.5.3 Test Suite Selection

Table 8.4 summarizes the test suite selection by showing the relationships between the archived test cases and the test suite partitions. Because the new command does not occur in the existing test cases, it is difficult to exploit the original test suites for regression tests. Therefore, the *Reuse* and *Modify* sets are empty. We concentrate on *test scripts* that contain the name of *scripting classes.* Test generation expands script classes. If the new command is a member of that scripting class, it will be generated

Table 8.4: Add a Command : Test Suite Selection and Modification Rules

Archive Type	Reuse	Mod	Regen
Test Script	$\emptyset$	$\emptyset$	Search for script class names where the new command is a member of the class.
Test Template	$\emptyset$	$\emptyset$	$\emptyset$
Test Case	$\emptyset$	$\emptyset$	$\emptyset$

Table 8.5: Add a Command - Regression Test Suite Construction Schemes

Regression Test Suite Source	RSD^v_{add-1} (Minimal)	RSD^v_{add-1}	RSD^v_{add-2} (Maximal)
Regeneration		X	X
New Test Cases	X	X	X

in the test case. Therefore, all test scripts that contain the name of a scripting class in which the new command is defined are placed into the *Regeneration* set.

8.5.4 Regression Test Suite Construction

Regression test suite construction combines tests from the *Regeneration* set and New Test Cases to create regression test suites. The choice of regression subdomain and the combinations of tests from these two sources determines how aggressive or conservative the regression test is. Table 8.5 shows three levels of regression test suites. A minimal test simply uses new tests from RSD^v_{add-1}. This provides an immediate regression test set for the new command. The second regression test suite uses tests from the regeneration set and new tests. Both use the RSD^v_{add-1} regression subdomain during test generation. A maximal regression suite uses regression subdomain RSD^v_{add-2} to regenerate original tests and to create new tests. This subdomain includes all components of the domain model influenced by the new command. Tests generated from this subdomain not only test the new command but also test how the new command influences other parts of the domain. Maximal regression tests combine tests from the previous version of the system under test with the functionality of the new command.

8.6 Modifying a Command - Deleting Part of the Command

8.6.1 Regression Domain Definition

Modifying a command is the most common change to a command language. The regression test suite generation rules for command modification could be realized by applying the **Delete Command** rules followed by the **Add Command** rules. A more refined approach is to define regression test suite selection rules for specific modifications. Two language modifications are presented here, *deleting part of a command* and *adding new parts to a command.* Consider deleting part of a command. The first step in the regression test process is to define, $RD^{v}_{mod\_del}$, by updating the domain model. The list below defines steps to update the domain model.

Script Class	If the deleted path removes functionality from the command, Delete the command name from the appropriate scripting classes.
Script Rule	If the deleted part of the command invalidates a script rule, Delete the Script Rule.
Command Syntax	Update command syntax by removing the deleted part of the command.
Intracommand Rules	If the deleted part of the command invalidates an ICR, Delete the Intracommand Rule.
Pre/Post Conditions	Inspect command pre/post conditions for update or removal.
Parameters	Parameters unique to the deleted part of the command are removed. For all parameters in the deleted part of the command, Inspect parameter constraint rules for update or removal.

8.6.2 Regression Subdomain Definition

The next step is to define *regression subdomains.* We list steps to define $RSD^{v}_{mod\_del-1}$ whose test criteria is to focus test generation on parts of the domain model influenced by deleting part of a command.

- Define Set P such that P contains all parameters of the modified command that are not unique to this command. Set P thus provides an indicator of the degree of interaction between the modified command and existing objects.
- Define Set C such that C contains all commands that use parameters in P. Set C defines all ways that existing commands use objects in common with the modified command.
- Define Set SC such that SC contains all script class names that contain commands in C.
- Define Set SR such that SR contains all script rules that use commands in C.

Table 8.6: Modify a Command (Delete Part of a Command): Test Suite Selection and Modification Rules

Archive Type	Reuse	Mod	Regen
Test Script	∅	∅	Contains the name of the modified command
Test Template	∅	∅	Regenerate new command using the new command syntax
Test Case	Contains the modified command, but does not contain the deleted part of the command	∅	Regenerate new commands using the new command syntax

8.6.3 Test Suite Selection

Table 8.6 summarizes test suite selection by showing how all three test archives are used as sources for regression testing. Test engineers scan test scripts for the name of the modified command. These scripts become part of the *Regeneration* set. They use the new command syntax, when regenerated during regression test suite construction, Test templates require a more thorough inspection. Testers examine each template for the name of the modified command. If the command is found and the template uses the deleted syntax, then the test template is placed in the *Regeneration* set. A new template is required during regeneration to pick up the new command syntax. Test cases are placed in the *Reuse* set if the test contains the modified command and the modified command does not use the deleted syntax. If the test case contains the modified command and it uses the deleted syntax, it is placed in the *Regeneration* set. During regression test suite construction, each modified command requires regeneration with new command syntax and semantics.

8.6.4 Regression Test Suite Construction

Regression test suite construction combines New Test Cases and tests from the Reuse, Modify, and Regeneration sets and to create regression test suites. Table 8.7 shows four levels of regression test suites ranging from a minimal test to a maximal test. A minimal regression test suite chooses tests from the Reuse set. This provides an

Table 8.7: Modify a Command (Delete Part of a Command): Regression Test Suite Construction Schemes

Regression Test Suite Source	$RSD^v_{mod\_del-1}$ (Minimal)	$RSD^v_{mod\_del-1}$	$RSD^v_{mod\_del-1}$ (Maximal)
Reuse	X	X	X
Regenerate		X	X
New Test Cases			X

immediate set of tests for the updated command language. A more conservative regression test includes tests from the Reuse and Regeneration sets. Sometimes there may be few tests in the Reuse and Regeneration sets. To fully test the modified command, we may need to build a regression test set from all sources.

8.7 Modifying a Command - Adding a New Part to a Command

8.7.1 Regression Domain Definition

Adding a new part to a command is common when new functionality is required or an existing command is updated to handle new hardware or software features. Regression domain definition for $RD^v_{mod\_add}$ is listed below.

Script Class	If the new path adds new functionality to the command, Add the command to the appropriate script classes.
Script Rule	Add script rules for the new command path.
Command Syntax	Update command syntax with the new path.
Intracommand Rules	Inspect old ICRs for update with the new command syntax. Add ICR's if the new syntax requires them.
Pre/Post Conditions	Inspect command pre/post conditions for update, additions, or removal.
Parameters	If new parameters are introduced by the new command syntax, Define new parameter value sets. For all parameters in the new command path, Inspect their parameter constraint rules for update, addition, or removal.

8.7.2 Regression Subdomain Definition

The regression subdomain $RSD^v_{mod\_add-1}$ defines test criteria to test the modified command its influence on the entire domain model. The steps listed below show how to construct the subdomain.

Table 8.8: Modify a Command (Adding a New Part to a Command): Test Suite Selection and Modification Rules

Archive Type	Reuse	Mod	Regen
Test Script	$\emptyset$	$\emptyset$	Contains the modified command
Test Template	$\emptyset$	$\emptyset$	Contains the modified command
Test Case	$\emptyset$	$\emptyset$	Contains the modified command

- Define Set P such that P contains all parameters of the modified command. Set P provides an indicator about the degree of interaction between the modified command and existing commands.
- Define Set C such that C contains all commands that use parameters in P. Set C defines all ways that existing commands use objects in common with the modified command.
- Define Set SC such that SC contains all script class names that contain commands in C.
- Define Set SR such that SR contains all script rules that use commands in C.

8.7.3 Test Suite Selection

Table 8.8 summarizes the test suite selection. Original tests can be a rich source for regression testing new paths in a command. Test scripts, test templates, and test cases are scanned for instances of the modified command. If a test contains the command, it is placed in the *Regeneration* set. Test scripts are lists of command names. During regeneration, each name is expanded into a test template and then into a test case. The current syntax and semantic rules for the modified command are used during regeneration. Test templates and test cases require a two step regeneration process. First, each occurrence of the modified command must re-generate its command template using the new syntax defined in the regression subdomain. In the second step, parameter values are selected for the updated command.

8.7.4 Regression Test Suite Construction

Regression test suite construction is summarized in Table 8.9. The source for regression test suites is limited to regenerating test cases and generating new test cases.

Table 8.9: Modify a Command (Adding a New Part to a Command): Regression Test Suite Construction Schemes

Regression Test Suite Source	$RSD^{v}_{mod\_add-1}$ (Minimal)	$RSD^{v}_{mod\_add-1}$ (Maximal)
Regeneration		X
New Test Cases	X	X

The minimal regression test chooses tests only by generating new test cases. The maximal regression combines tests generated from the regeneration suite and from new test cases. Regression test generation for this modification seems limited, but this can be expected when new functionality requires new commands or modifying existing commands with new paths or syntax. Because there is new syntax, parameters, and semantics associated with the modification, the regression test construction relies on creating new test cases.

8.8 Summary

Regression testing is one approach to retesting software after change, update, or modification. One source of tests for regression testing is the set of original test cases. Many times, *all* of the tests cannot be rerun. Instead, the tester must selectively choose tests from the original set along with generating new tests to adequately test a software modification. In this chapter, we presented Domain Based Regression Testing (DBRT) where changes to the system under test are translated into changes in the domain model. The modified domain model is called the regression domain. Test criteria defined for the regression test is captured in a regression subdomain. We also examine the set of original tests for use in the regression test suite. The tests are partitioned into four sets: Discard, Reuse, Modify, and Regeneration. Testers also generate new tests based on the regression subdomain. Regression test suite construction combines tests from these sources to create a variety of regression tests. We detailed DBRT by defining regression testing rules for four command language modifications.

Chapter 9

RESEARCH CONTRIBUTIONS AND FUTURE WORK

9.1 Contributions

This research examined automated test data generation for the system level test of command-based applications. We pursued this topic to generalize system test by viewing an application through its user interface. The solution required coupling three components: (1) Abstract representation of the command language, (2) Test generation based on the abstract representation, and (3) Regression test support based on the representation and integrated into the test generation process. The result is a test generation method called Domain Based Testing (DBT). At its core, DBT relies on a domain model representation of command language syntax and semantics. This is the first time domain modeling techniques have been used as a basis for automated test generation. The results are promising.

- *Abstract Representation of a Command Language*

The domain model forms a kernel for a flexible and uniform approach to test data generation. Domain analysis for command-based systems specifies how to analyze a command language for testing purposes. The resulting domain model captures command language syntax and semantics for command languages.

- *Test Generation Process*

The DBT test generation process uses the domain model for test generation and it extends the domain model representation by mapping test criteria into test subdomains. Test subdomains can represent a variety of test criteria and support multiple test criteria in a single subdomain. Test subdomains can be general purpose or narrowly defined. The result is a uniform approach to test generation because the test generator uses the test subdomain (i.e., domain model and test criteria combination) as input to the test generation process.

- *Regression Testing*

One objective for this research is to support regression testing. Regression testing is one way to extend a test data generation method to the maintenance phase of the software life cycle. We developed Domain Based Regression Testing (DBRT) as part of the DBT method. DBRT extends the domain model approach to test generation by mapping changes in the command language to changes in the domain model. The result is a consistent, uniform, and flexible approach to testing evolutionary software products.

- *Automated Tool Support*

We applied DBT concepts to an automated test generation tool called *Sleuth*. *Sleuth* follows the DBT test generation process, provides tools to define domain models, contains utilities to configure test subdomains, and offers a simple interface to generate, archive, and recall test cases. We can test the following features of a command language using *Sleuth*: parser, command sequencing, parameter values, parameter constraints, and system objects. *Sleuth* also supports test generation based on operational profiles. *Sleuth* has been used as an experimental tool and in an industrial testing group. Its success serves as a "proof-of-concept" for the DBT method.

- *Test Effectiveness Prediction*

This research shows how to use a neural network to predict test case effectiveness. This increases test efficiency by running tests likely to expose faults and eliminating tests that do not predict fault exposure. We emphasize that the contribution of these results should be not restricted to Domain Based Testing. Even though neural net predictors were applied to our research, they can be used by other test data generation methods.

- *Test Generation based on AI Planning*

This research also shows how to use an AI Planning system as an automated test generator. This is the first time a planner has been used in this role. Testers view

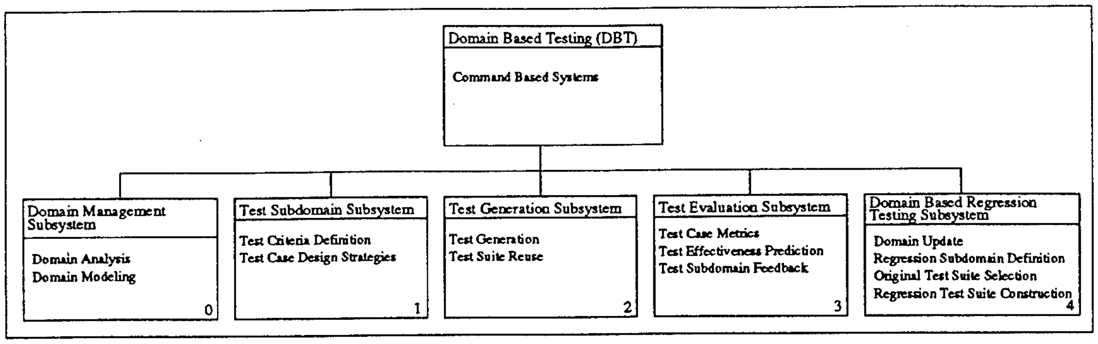

Domain Based Testing (DBT)
Command Based Systems
Domain Management Subsystem
Domain Analysis
Domain Modeling
0
Test Subdomain Subsystem
Test Criteria Definition
Test Case Design Strategies
1
Test Generation Subsystem
Test Generation
Test Suite Reuse
2
Test Evaluation Subsystem
Test Case Metrics
Test Effectiveness Prediction
Test Subdomain Feedback
3
Domain Based Regression Testing Subsystem
Domain Update
Regression Subdomain Definition
Original Test Suite Selection
Regression Test Suite Construction
4

Figure 9.1: Domain Based Testing Architecture

test generation differently when using a planner. They focus on the goals of test case generation and they leave the choice of commands, command sequences, and parameters to the planner. The results are tests that achieve the test generation goal. These tests can involve innovative command sequences that may not be considered by the testers.

9.2 Future Work

The Domain Based Testing (DBT) architecture defined in this dissertation establishes a structure for continued research (see Figure 9.1). We seek to improve DBT, add new features to the existing DBT model, and propose exploratory research for the DBT subsystems. Table 9.1 lists five area for future work. Each is detailed below:

- *Apply DBT to Other Command-Based Systems*

First, we need to apply DBT to other command-based systems. This will evaluate our claim of DBT as a general purpose approach for system level testing of an application through its command language user interface. The current domain model development was guided by the StorageTek HSC command language. Applying DBT to other application domains will support our current model or it will guide us on ways to improve it.

- *Domain Model Representation*

The second research topic suggests investigation of different ways to represent domain model components. Most important are extensions to the script rule representation and extensions to parameter value set definitions. The current hybrid test generation

Table 9.1: Future Work - Domain Based Testing

DBT Topic	Future Work
Application Domains	Apply DBT to other application domains.
Domain Model Representation for Command Based Systems	Script Rules. Parameter Value Set Language.
Test Generation Engines for Command Based Systems	Experiment with Hybrid Implementation. Full Hybrid Implementation. Extend AI Planner. Evaluate state information required during test generation.
Domain Based Regression Testing	Formal specification for DBRT. Develop DBRT tools and incorporate into *Sleuth*
Test Case Effectiveness	Refine metrics/attributes to use as input to the neural network. Show neural net prediction using actual test case data. Show the adaptive nature of the neural network. Use neural network predictions to derive test subdomains.

engine uses *macro expansion* to represent script rules. Experiments with *Sleuth* show that macro expansion provides useful support for script rules. For example, the `MOUNT <5/any> DISMOUNT` sequence specifies a rule for mount-dismount pairs and it defines parameter binding between the `MOUNT` command and the `DISMOUNT` command. We need a more powerful scripting rule representation to capture arbitrary command sequences. We also need to support parameter binding to commands other than the first and last command in the macro expansion. We recommend a *regular expression* representation to increase scripting rule power.

We also recommend enhancements to Parameter Value Definition. These improvements should unify parameter value definition, parameter constraint rules, and test criteria support. The current parameter set definition supports these features directly or with some coaxing. Refining parameter value definition would improve automated tool support, improve domain capture, and increase test generation efficiency. We propose a *set language* to refine the parameter value definition. The set language could be used to make domain capture easier (at the parameter level). Testers could use the same set language to define parameter constraint rules and record test criteria. The test generator would use the set language to automatically generate test subdomain definitions such as *invalid* or *boundary-value* parameter sets. If we include set types such as *ordered sets* and *unordered sets*, we can provide additional test criteria support. For instance, the notion of an ordered set makes it possible to sequentially choose parameter values. Ordered and unordered sets will make it possible to choose parameter values similar to test generation methods like *category-partition* testing [OB88].

- *Test Generation Engines*

Our third topic for future work includes improvements to the test generation tools, *Sleuth* and the AI Planner. We recommend a full implementation of the test generation process as defined in this dissertation. For instance, *Sleuth* provides editors and utilities to define test subdomains. These tools are not fully automated but some features could be. For instance, this research shows how to "automatically" modify script rules, parameter binding rules, and command syntax using look-up tables. *Sleuth* requires the tester to manually configure all three of these test subdomain components. We also recommend experimental extensions to the hybrid test generator. The hybrid generator uses a collection of sentence generation tools and algorithms to create test cases. The beauty of the hybrid approach is its adaptability to new algorithms and test generation methods. We encourage experimental research into new and better test generation algorithms and tools. For instance, ELI and DGL are language translation tools. ELI is based on attribute grammars [Wai93, Kas93, Kas91], and DGL is based on a probabilistic context-free grammar [Mau94]. If we consider the three stage test generation sequence as three *translation* problems, tools like ELI or DGL may be appropriate for one or more of these stages.

We also recommend a full investigation of the AI Planner approach to test data generation. Our experiments show how to use a planner as a test generator for command-based systems. The next step is to extend the planning system to include a full domain model, improve its runtime performance, and compare its test case effectiveness with the hybrid method. During the planning research, we also identified an open research question. How much state information is required by an automated test generator? If we include "all" state information, then the test generator becomes a *simulation* of the system under test. We do not intend to build a simulator. Instead, we want to use as little state information as possible while maintaining high test case effectiveness. A full implementation of the AI Planner will address the argument for complete state information. Yet, in some command languages, we may not have complete state information at our disposal. Sometimes the effects of a command may be conditional on unknown factors, probabilistic, or simply unknown. We might be able to relax the requirements for complete state information by studying "planning under uncertainty." In a related experiment, we could investigate adding more state information to the hybrid test generator. Currently, *Sleuth* uses as little state information as possible. We recommend adding *state parameter* information to *Sleuth* as a first step. A pilot study and comparison of tests generated with and without state information could resolve these questions and its usefulness in an automated test generator. If the results are favorable for including state information, additional state information from *nonparameter state* and *nonparameter events* could be added to *Sleuth*. Nonparameter object elements add several concerns for the DBT test generation process: (1) How do we represent nonparameter state and events?, (2) What test generation stage resolves the nonparameter information, and (3) What test generation algorithms must be updated to incorporate the added domain information?

- *Domain Based Regression Testing*

The fourth research topic suggests extensions to and experiments with Domain Based Regression Testing (DBRT). First, testers need to field test the DBRT specification. This experiment will refine the DBRT method, improve the process, and clarify the regression test approach. During the field test, we need to consider the application of multiple command language changes into one regression domain. The current DBRT specification shows how to test command language changes individually. Real-world regression testing may require testing of multiple command language modifications simultaneously. Results from testing multiple command language modifications may require changes to the DBRT specification. After the field test and subsequent refinement of the DBRT specification, DBRT needs a process model. The DBRT Process could use the Domain Analysis Process Model in Chapter 4 as a foundation. Many of the editors and services defined for the Domain Analysis Process could be used by the regression testing tools. We know that the regression testing process will use many of the ADTs and the Domain Management Services to access existing domain models and to create regression domains. After this investigation, we need to develop tool support for DBRT. One way to accomplish this is to add regression test generation tools to *Sleuth.* We may not be able to automate the entire regression test process, but tool support is necessary for increased tester productivity and decreased test data generation time.

- *Test Case Effectiveness*

Finally, we demonstrated the use of a neural network to predict test case effectiveness. The next logical steps include four related research topics. First, we need to examine a broader set of test case attributes for neural net training. For our tests, we selected metrics and attributes as part of a controlled experiment. The question remains, What metrics/attributes are best to predict test case effectiveness? Perhaps a variety of domain coverage measures and attributes calculated from a test case should be included during *neural net training.* After training, we can prune those that are not significant fault indicators. Because fault indicators vary based on command language, application under test, and software maturity, a two stage train-prune approach may be needed to generalize the neural net predictor. Second, we need a study of neural net prediction using actual test case data. Our test oracle and controlled experiment shows proof-of-concept, but we need to evaluate a neural net in the field. The third topic for neural net research investigates an evaluation of the adaptive capabilities of a neural net classifier. As software matures the fault indicators are likely to change. A neural net initially trained for fault prediction may need retraining as the software matures to maintain good predictions. Finally, we see a potential use for the neural net predictor as a feedback mechanism in the test generation process. Running a neural network "backwards" will identify test metrics required to produce particular faults. This information could drive test subdomain definition. Figure 9.2 shows how to include the feedback mechanism into the test generation process.

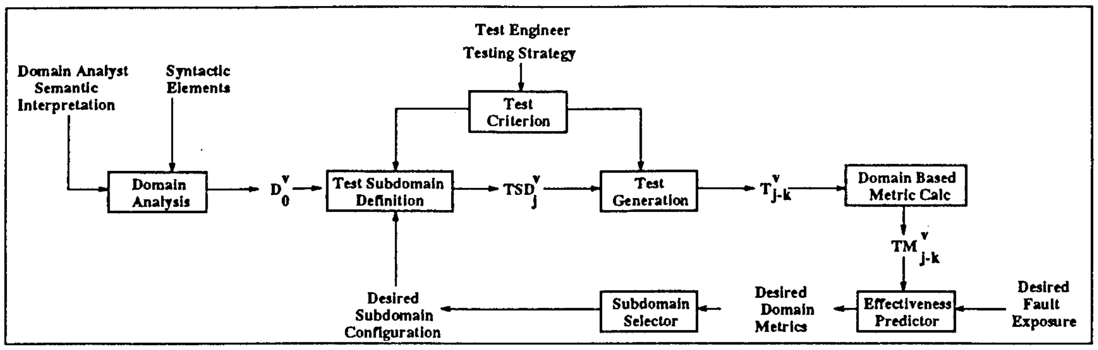


Figure 9.2: Test Generation Process Model with Subdomain Selection Based on a Fault Predictor

9.3 Summary

The goal of this research was to develop a general purpose test generation approach for the system level test of an application. We achieved this goal by viewing a system through its command language user interface. We required an abstract representation of command language syntax and semantics and we needed a test generation method based on this representation. Our solution, called Domain Based Testing (DBT), uses a domain model representation. All test data generation, test criteria, and regression testing support rely on the domain model. The result is a new approach to system level testing of a command-based application. We demonstrated that it is a sound test generation method. Experiments and actual production use serves as a "proof-of-concept" for DBT. During this research, we also uncovered new topics to investigate, alternatives to evaluate, and ideas for future research.

REFERENCES

[ABC82] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, Verification, and Testing of Computer Software. *Computing Surveys*, 14(2):159–192, June 1982.

[BCC88] P. Benedusi, A. Cimitile, and U. De Carlini. Post-maintenance Testing Based on Path Change Analysis. In *Proceedings of the Conference on Software Maintenance*, pages 352–361, 1988.

[Bei90] Boris Beizer. *Software Testing Techniques*. VanNostrand Reinhold, second edition, 1990.

[BF79] Jonathan A. Bauer and Alan B. Finger. Test Plan Generation Using Formal Grammars. In *Proceedings of the Fourth International Conference on Software Engineering*, pages 425–432, 1979.

[BGPW93] Anthony Barrett, Keith Golden, Scott Penberthy, and Daniel Weld. *UCPOP* User's Manual. Technical Report TR 93-09-06, Dept of Computer Science and Engineering, University of Washington, Seattle, 1993.

[Big92] Ted J. Biggerstaff. *Advances in Computers*, chapter An Assessment and Analysis of Software Reuse. Academic Press, 1992.

[Boo83] Grady Booch. *Software Engineering with Ada*. Benjamin/Cummings: Menlo Park, CA, 1983.

[Boo87] Grady Booch. *Software Components with Ada: Structures, Tools, and Subsystems*. Benjamin/Cummings: Menlo Park, CA, 1987.

[Boo91] Grady Booch. *Object Oriented Design with Applications*. Benjamin/Cummings: Menlo Park, CA, 1991.

[BP89] Ted J. Biggerstaff and Alan J. Perlis, editors. *Software Reusability - Concepts and Models*, volume I of *Frontier Series*. ACM Press, 1989.

[Bra92] Marshall Brain. *Motif Programming: The Essentials and More*. Digital Press, 1992.

[BS82] Franco Bazzichi and Ippolito Spadafora. An Automatic Generator for Compiler Testing. *IEEE Transactions on Software Engineering*, 8(4):343–353, 1982.

[CB92] Chye-Lin Chee and Jit Biswas. Experience with Integrating Operating Systems. In Peter A. Ng C.V. Ramamoorthy Laurence C. Seifert and Raymond T. Yeh, editors, *Proceedings of the Second International Conference on Systems Integration*, page 593, 1992.

[CDK+89] B.J. Choi, R.A. DeMillo, E.W. Krauser, R.J. Martin, A.P. Mathur, A.J. Offutt, H.Pan, and E.H. Spafford. The Mothra Tool Set. In *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, volume II, pages 275–284, 1989.

[CF82] Paul R. Cohen and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence*, volume 3. HeurisTech Press, 1982.

[CFR90] James Collofello, Terry Fisher, and Mary Rees. A Testing Methodology Framework. In George J. Knafl, editor, *Proceedings of the IEEE 14th Annual International Software and Applications Conference*, pages 577–586, 1990.

[Cho77] Tsum S. Chow. Testing Software Design Modeled by Finite State Machines. In *Proceedings of the First COMPSAC*, pages 58–64, 1977.

[CPRZ89] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A Formal Evaluation of Data Flow Path Selection Criteria. *IEEE Transactions on Software Engineering*, 15(11):1318–1332, November 1989.

[Cra93] Stewart Crawford. Prototype for Storage Tek Automated Test Generator. Internal Report, 1993.

[CRV+80] A. Celentano, S. Crespi Reghizzi, P. Della Vigna, C. Ghezzi, G. Gramata, and F. Savoretti. Compiler Testing using a Sentence Generator. *Software-Practice and Experience*, 10:897–918, 1980.

[CY90] Peter Coad and Edward Yourdon. *Object-Oriented Analysis.* Computing Series. Yourdon Press : Englewood Cliffs, NJ, second edition, 1990.

[DH81] A.G. Duncan and J.S. Hutchison. Using Attributed Grammars to Test Designs and Implementations. In *Proceedings of the Fifth International Conference on Software Engineering*, pages 170–177, 1981.

[DO91] Richard A. DeMillo and A. Jefferson Offutt. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[FA88] Stephen Fickas and John Anderson. A proposed perspective shift: Viewing specification design as a planning problem. Technical report, Department of Computer and Information Science, University of Oregon, Eugene, OR, November 1988.

[Fau94] Laurene Fausett. *Fundamentals of Neural Networks.* Prentice Hall:Englewood Cliffs, New Jersey, 1994.

[Fis77] Kurt F. Fisher. A Test Case Selection Method for the Validation of Software Maintenance Modifications. In *Proceedings of the International Computer Software and Application Conference*, pages 421–426, 1977.

[FW88] Phyllis G. Frankl and Elaine J. Weyuker. An Applicable Family of Data Flow Testing Criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[FW93a] P.G. Frankl and S.N. Weiss. An Experimental Comparison of the Effectivenss of Branch Testing. *IEEE Transactions on Software Engineering*, SE-19(8):774–787, August 1993.

[FW93b] P.G. Frankl and E. J. Weyuker. A Formal Analysis of Fault-Detecting Ability of Testing Methods. *IEEE Transactions on Software Engineering*, SE19(3):202–213, March 1993.

[FW93c] P.G. Frankl and E.J. Weyuker. Provable Improvements on Branch Testing. *IEEE Transactions on Software Engineering*, SE-19(10):962–975, October 1993.

[GH88] David Gelperin and Bill Hetzel. The Growth of Software Testing. *Communications of the ACM*, 31(6):687–695, June 1988.

[Gom91] Hassan Gomaa. An Object-Oriented Domain Analysis and Modeling Method for Software Reuse. In V. Milutinovic B.D. Shriver J.F. Nunamaker Jr. and R.H. Spague Jr., editors, *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, volume 2, pages 46–56, 1991.

[Ham89] Richard Hamlet. Theoretical Comparison of Testing Methods. In *Proceedings of the Third Symposium on Testing, Analysis, and Verification*, pages 28–37, 1989.

[HC91] James W. Hooper and Rowena O. Chester. *Software Reuse - Guidelines and Methods.* Plenum Press, 1991.

[Het84] William Hetzel. *The Complete Guide to Software Testing.* QED Information Sciences, 1984.

[How85] William E. Howden. The Theory and Practice of Functional Testing. *IEEE Software*, pages 6–17, September 1985.

[How86] William E. Howden. A Functional Approach to Program Testing and Analysis. *IEEE Transactions on Software Engineering*, 12(10):997–1005, October 1986.

[How87] William E. Howden. *Functional Program Testing and Analysis.* McGraw-Hill, 1987.

[How89] William E. Howden. Validating Programs without Specifications. In Richard A. Kemmerer, editor, *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 2–8, 1989.

[HS88] M.J. Harrold and M.L. Soffa. An Incremental Approach to Unit Testing During Maintenance. In *Proceedings Conference on Software Maintenance*, pages 363–367, 1988.

[HS89a] M.J. Harrold and M.L. Soffa. An Incremental Data Flow Testing Tool. In *Proceedings of the Sixth International Conference on Testing Computer Software*, 1989.

[HS89b] M.J. Harrold and M.L. Soffa. Interprocedural Data Flow Testing. In *Proceedings of the Third Symposium on Software Testing, Analysis, and Verification*, pages 158–167, 1989.

[HT90] Dick Hamlet and Ross Taylor. Partition Testing Does Not Inspire Confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, December 1990.

[Huf92] Karen Huff. Software Adaptation. *Working Notes of AAAI-92 Spring Symposium on Computational Considerations in Supporting Incremental Modification and Reuse*, pages 63–66, March 1992.

[Jeo92] Taewoong Jeon. *A Knowledge-Based System for Regression Testing.* PhD thesis, Illinois Institute of Technology, May 1992.

[Kas91] U. Kastens. *Attribute Grammars in a Compiler Construction Environment*, volume 545 of *Lecture Notes in Computer Science.* Springer, 1991.

[Kas93] U. Kastens. *Executable Specifications for Language Implementation*, volume 714 of *Lecture Notes in Computer Science.* Springer, 1993.

[Kru92] Charles Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[LG89] J.R. Lyle and K.B. Gallagher. A Program Decomposition Scheme with Applications to Software Modification and Testing. In *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, volume 2, pages 479–485, 1989.

[LK83] J. W. Laski and B. Korel. A Data Flow Oriented Program Testing Strategy. *IEEE Transactions on Software Engineering*, SE-9(3):347–354, May 1983.

[LW89] Hareton K.N. Leung and Lee White. Insights into Regression Testing. In *Proceedings - Conference on Software Maintenance 89*, pages 60–69, October 1989.

[LW91] Hareton K.N. Leung and Lee White. A Cost Model to Compare Regression Test Strategies. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 201–208, 1991.

[Mau94] Peter M. Maurer. Reference Manual for a Data Generation Language Based on Probabilistic Context Free Grammars. Technical report, University of South Florida, Tampa, 1994.

[Mos93] Daniel J. Mosley. *The Handbook of MIS Application Software Testing: Methods, Techniques, and Tools for Assuring Quality Through Testing.* Computing Series. Prentice-Hall : Englewood Cliffs, NJ, 1993.

[MRtPRG86] J.L. McClelland, D.E. Rumelhart, and the PDP Research Group. *Parallel Distributed Processing: Exploration in the Microstructure of Cognition*, volume 1. MIT Press : Boston, 1986.

[Mul89] Mark Mullin. *Object Oriented Program Design with Examples in C++.* Addison-Wesley, 1989.

[Mun88] Carlos Urias Munoz. An Approach to Software Product Testing. *IEEE Transactions on Software Engineering*, 14(11):1589–1596, November 1988.

[Mye76] Glenford J. Myers. *Software Reliability : Principles and Practices.* John Wiley and Sons, 1976.

[Mye79] Glenford J. Myers. *The Art of Software Testing.* John Wiley and Sons, 1979.

[Nta84a] Simeon C. Ntafos. An Evaluation of Required Element Testing Strategies. In *Proceedings of the 7th Conference on Software Engineering*, pages 250–256, 1984.

[Nta84b] Simeon C. Ntafos. On Required Element Testing. *IEEE Transactions on Software Engineering*, 10(6):795–803, November 1984.

[Nta88] Simeon C. Ntafos. A Comparison of Some Structural Testing Strategies. *IEEE Transactions on Software Engineering*, 14(6):868, June 1988.

[OB88] Thomas J. Ostrand and Marc J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[OM91] A.A. Omar and F.A. Mohammed. A Survey of Software Functional Testing Methods. *ACM SIGSOFT Software Engineering Notes*, pages 75–82, April 1991.

[O'r93] O'reilly, editor. *Motif Reference Manual for OSF/Motif Release 1.2*, volume Volume 6B. O'reilly, motif edition, 1993.

[Pay78] A.J. Payne. A Formalized Technique for Expressing Compiler Exercises. *ACM SIGPLAN Notices*, 13:59–69, 1978.

[Per86] William E. Perry. *How to Test Software Packages.* John Wiley and Sons, 1986.

[Pet85] Nathan H. Petschenik. Practical Priorities in System Testing. *IEEE Software*, pages 18–23, September 1985.

[Pur72] Paul Purdom. A Sentence Generator for Testing Parsers. *BIT*, 12(3):366–375, 1972.

[PZ91] A. Parrish and S.H. Zweben. Analysis and Refinement of Software Test Data Adequacy Properties. *IEEE Transactions on Software Engineering*, SE-17(6):565–581, June 1991.

[RAO89] Debra J. Richardson, Stephanie Leif Aha, and Leon J. Osterweil. Integrating Testing Techniques Through Process Programming. In *Proceedings of the ACM SIGSOFT89 - Symposium on Software Testing, Analysis, and Verification*, December 1989.

[RC81] Debra J. Richardson and Lori A. Clarke. A Partition Analysis Method to Increase Program Reliability. In *Proceedings of the Fifth International Conference on Software Engineering*, pages 244–245, 1981.

[RG92] Kenneth S. Rubin and Adele Goldberg. Object Behavior Analysis. *Communications of the ACM*, 35(9):48–62, September 1992.

[ROT89] Debra J. Richardson, Owen O'Malley, and Cindy Tittle. Approaches to Specification-Based Testing. In Richard Kemmer, editor, *Proceedings of the Third Symposium of Software Testing Analysis and Verification*, pages 86–96. ACM SIGSOFT, Association of Computing Machinery, December 1989.

[RW82] S. Rapps and E. J. Weyuker. Data Flow Analysis Techniques for Test Data Selection. In *Proceedings of the Sixth International Conference on Software Engineering*, pages 272–277, September 1982.

[RW85] S. Rapps and E. J. Weyuker. Selecting Software Test Data using Data Flow Information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[Sne93] Harry M. Sneed. Regression Testing in Reengineering Projects. In *International Conference on Program Cognition*, June 1993.

[ST80] Robert G.D. Steel and James H. Torrie. *Principles and Procedures of Statistics: A Biometrical Approach.* McGraw-Hill : New York, second edition, 1980.

[TCY93] Will Tracz, Lou Coglianese, and Patrick Young. A Domain-Specific Software Architecture Engineering Process Outline. *ACM SIGSOFT Software Engineering Notes*, 18(2):40–49, April 1993.

[TDN93] Markos Z. Tsoukalas, Joe W. Duran, and Simon Ntafos. On Some Reliability Estimation Problems in Random and Partition Testing. *IEEE Transactions on Software Engineering*, 19(7):687–697, July 1993.

[Tek92] Storage Tek. *StorageTek 4400 Operator's Guide. Host Software Component (VM) Rel 1.2.0.* StorageTek, 1992.

[Tra92] Will Tracz. Domain Analysis Working Group - First International Workshop on Software Reusability. *ACM SIGSOFT Software Engineering Notes*, 17(3):27–34, July 1992.

[vM90] Anneliese von Mayrhauser. *Software Engineering - Methods and Management.* Academic Press: Boston, MA, 1990.

[vM93] Anneliese von Mayrhauser. Telephone Conversation with J.S. Hutchison. Telecon, September 1993.

[vMAM95] Anneliese von Mayrhauser, Charles Anderson, and Richard Mraz. Using a Neural Network to Predict Test Case Effectiveness. In *Proceedings of the 1995 IEEE Aerospace Applications Conference.* IEEE, February 1995.

[vMJ93] Anneliese von Mayrhauser and Taewoong Jeon. CASE Tool Architecture for Knowledge-Based Regression Testing. In *Tri-Ada93*, 1993.

[vMO93] Anneliese von Mayrhauser and Kurt Olender. Efficient Testing of Software Modifications. In *International Conference on Testing*, 1993.

[Wai93] William M. Waite. An Executable Language Definition. In *ACM SIGPLAN Notices*, volume 28. ACM, February 1993.

[Wal94] Jeff Walls. *Sleuth* : An Automated Test Generation Tool. Master's thesis, Colorado State University, May 1994.

[WC80] Lee White and Edward Cohen. A Domain Strategy for Computer Programming Testing. *IEEE Transactions on Software Engineering*, 6(3):247–257, May 1980.

[Wey86] E. Weyuker. Axiomatizing Software Test Data Adequacy. *IEEE Transactions on Software Engineering*, SE-12(12):1128–1138, December 1986.

[Wey88] E. Weyuker. The Evaluation of Program-Based Software Test Data Adequacy Criteria. *Communications of the ACM*, 31(6):668–675, June 1988.

[Wey90] Elaine J. Weyuker. The Cost of Data Flow Testing: An Empirical Study. *IEEE Transactions on Software Engineering*, 16(2):121–128, February 1990.

[Wey93a] E.J. Weyuker. Can We Measure Software Testing Effectiveness. In *Proceedings of the International Software Metrics Symposium*, pages 100–107. IEEE Computer Society, May 1993.

[Wey93b] E.J. Weyuker. More Experience with Data Flow Testing. *IEEE Transactions on Software Engineering*, SE-19(9):912–919, September 1993.

[WHH80] M. Woodward, R. Hedley, and M.A. Hennel. Experience with Path Analysis and Testing of Programs. *IEEE Transactions on Software Engineering*, SE-6(3):278–286, May 1980.

[WJ91] Elaine J. Weyuker and Bingchiang Jeng. Analyzing Partition Testing Strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, July 1991.

[WL92] Lee J. White and Hareton K.N. Leung. A Firewall Concept for both Control-Flow and Data-flow in Regression Integration Testing. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 262–271, 1992.

[WO82] Cindy Wilson and Leon J. Osterweil. Omega - A Data Flow Analysis Tool for the C Programming Language. In *Proceedings of the Sixth International Computer Software Applications Conference*, pages 9–18. IEEE Computer Society, Computer Society Press, November 1982.

[WWH91] E.J. Weyuker, S.N. Weiss, and R. Hamlet. Comparison of Program Testing Strategies. In *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, pages 1–10, 1991.

[YC80] S.S. Yau and J. Collofello. Some Stability Measures for Software Maintenance. *IEEE Transactions on Software Engineering*, SE-6(6):545–552, November 1980.

[YK87] S.S. Yau and Z. Kishimoto. A Method for Revalidating Modified Programs in the Maintenance Phase. In *Proceedings of COMPSAC '87*, pages 272–277, 1987.

[ZG89] S.H. Zweben and J. Gourlay. On the Adequacy of Weyuker's Test Data Adequacy Criteria. *IEEE Transactions on Software Engineering*, SE-15(4):496–500, April 1989.

[Zur92] Jacek M. Zurada. *Introduction to Artificial Neural Systems.* West Publishing : St.Paul, 1992.

Appendix A

STORAGETEK HSC RELEASE 1.2 DOMAIN

A.1 Overview

The following tables, glossaries, and description define the HSC 1.2 Domain Model used throughout this research. It is presented so other researchers may continue this research, find better ways to represent the domain model components, and to capture the first problem used to demonstrate DBT principles.

A.2 Command Definition

Table A.1 lists all command names with a short description of their function.

Table A.1: HSC Release 1.2 Command Descriptions

Command Name	Description
ALLOC	Changes the Host Software Component (HSC) allocation options.
CAPPref	Assigns a preference value to one or more cartridge access ports (CAPs)
CDs	Enable / Disable copies of the control data set
CLean	Schedules the cleaning cartridge to be mounted on a library controlled transport
DISMount	Directs the Library Storage Module (LSM) to dismount a cartridge
DRAin	Terminates and ENter command
EJect	Directs the robot to take cartridges from a Library Storage Module (LSM) and places them into a cartridge access port (CAP) where they can be removed by an operator
ENter	Used to place cartridges into a Library Storage Module (LSM) through a cartridge access port (CAP) while operating in automatic model
Journal	Used to establish the action taken by the Host Software Component (HSC) if both journals fill to capacity before a control data set backup or a journal off-load is executed
LOad	Used to query the status of the current tape transport activity
MNTD	Set options on how the Host Software Component (HSC) processes the mounting and dismounting of library volumes
MODify	Places a Library Storage Module (LSM) online or offline to all hosts
MONITOR	Initiates monitoring of cartridge move requests from the programmatic interface
Mount	Directs the robot to mount a volume onto a specified library controlled transport
MOVe	Directs the robot to move cartridges to selected destinations within the same Library Storage Module (LSM) or to any LSM within an Automated Cartridge System (ACS)
OPTion	Used to set or change general purpose options of the HSC
RECover	Allows the operator to recover the resources owned by a host that becomes inoperable
RELease	Used to free an allocated cartridge access port (CAP)
RETry	Applies only to the JES3 environment. It enables the user to restart HSC/JES3 initialization without restarting the HSC address space component
SCRparm	Dynamically modifies the scratch warning thresholds and interval values for the host on which the command is issued
SENter	Used to schedule the enter of a single cartridge using a cartridge access port (CAP) that is currently allocated for ejecting cartridges
SET	Used to activate / deactivate various functions within the HSC
SRVlev	Used to specify the service level at which the Host Software Component (HSC) operates
STOPMN	Terminates the monitoring of cartridge move requests received from the programmatic interface
SWitch	Used in dual Library Management Unit (LMU) configuration to reverse the roles of the master and standby LMUs
TRace	Enables / Disables tracing of events for selected Host Software Components (HSCs)
UEXIT	Permits you to invoke your own processing routines at particular points during HSC processing
Vary	Places physical Library Management Unit (LMU) stations online, offline, or standby
VIew	If video monitors are attached to the LSM, the VIew command enables the operator to visually inspect internal components of the LSM using the robot's cameras
Warn	Used to establish the scratch warning threshold values

The next series of tables define command syntax and semantic rules. We use a BNF for this appendix because it is not convenient to show the syntax diagrams for each command. The following symbols are used in the BNF.

Symbol	Meaning
::=	Production Symbol
`foo Bar FRAMUS`	Terminal {`typewritten font`}
{`a` | `b` | `c`}	Make a choice of a, b, or c
[ ]	Option
<xyz>	Command Language Parameter
NONTERMINAL	Nonterminal {**BOLD CAPS**}

ALLOC	
Preconditions	• lmu-status(lmu-id) = ONLINE
Postconditions	
BNF	**alloc-cmd** `::= ALLOC` **ALLOC-CHOICE** `[,` **ALLOC-CHOICE**`]` `[<host-id>]` **ALLOC-CHOICE** `::= {`**DEFER** | **GDGALL** | **SPECVOL** | **UNITAFF** | **ZEROSCR**`}` **DEFER** `::= Defer({OFf | ON | JEs3 })` **GDGALL** `::= Gdgall( {NOSep | SEP})` **SPECVOL** `::= Specvol [(<acs-id>) | (<acs-range>) |` `(<acs-list>)]` **UNITAFF** `::= Unitaff({NOSep | SEP})` **ZEROSCR** `::= Zeroscr({OFf | ON)}`

CAPPref	
Preconditions	
Postconditions	preference(cap-id) = prefvalue
BNF	**cappref-cmd** `::= <prefvalue> [ 000 |` **CAP**`] [<host-id>]` **CAP** `::= {<cap-id> | <cap-range> | <cap-range> |` `(<cap-list>)]`

CDS	
Preconditions	• lmu-status(lmu-id) = ONLINE
Postconditions	• CDS-state(dsn) = {Enable | Disable}
BNF	**cds-cmd** `::= CDs {`**ENABLE** | **DISABLE**`}` **ENABLE** `::= Enable DSn=<dsn>` **DISABLE** `::= Disable {DSn=<dsn> | Primary | SEcndry | STandby }`

CLean	
Preconditions	• service-level = FULL • lmu-status(lmu-id) = ONLINE • lsm-status(lsm-id) = ONLINE • autoclean = ON
Postconditions	
BNF	**clean-cmd** `::= CLean {<drive-id>|<drive-range>|(<drive-list>)}` `[<host-id>]`

Dismount	
Preconditions	• service-level = FULL • lmu-status(lmu-id) = ONLINE • lsm-status(lsm-id) = ONLINE
Postconditions	
BNF	**dismount-cmd** `::= DISMount { , | <volser> } <drive-id>` `[<host-id>]`

Display	
Preconditions	• lmu-status(lmu-id) = ONLINE
Postconditions	
BNF	**display-cmd** `::= Display` [**ACS** | **STATUS** | **COMMAND** | **COMMPATH** | **LSM** | **MSG** | **MONITOR** | **SCRATCH** | **THRESHOLD** | **VOLSER**] **ACS** `::= Acs[<acs-id> | <acs-range> | (<acs-list>)]` **STATUS** `::= {All | ALLOC | Cap | CDS | MNTD | OPTion | SRVlev}` **COMMAND** `::= { CMd | COmmand } <command-name>` **COMMPATH** `::= COMMPath [HOSTid [ =* |=ALL|=<host-id>|=(<host-list>)] ]` **LSM** `::= Lsm [<lsm-id> | <lsm-range> | (<lsm-list>)]` **MSG** `::= {Msg | Message} <msg-id>` **MONITOR** `::= MONitor [,PGMI] [,L= [<cc> | <console-name>]]` **SCRATCH** `::= SCRatch [<acs-id> | <lsm-id>] [SUBpool(<subpool-name>)]` **THRESHOLD** `::= THREShld[<acs-id> | <lsm-id>] [SUBpool(<subpool-name>)]` **VOLSER** `::= {Volser | Volume} {<volser> | <vol-range> | (<vol-list>)} [DEtail]`

Drain	
Preconditions	• service-level = FULL • lmu-status(lmu-id) = ONLINE • lsm-status(lsm-id) = ONLINE
Postconditions	• cap-status(cap-id) = DRAINED
BNF	**drain-cmd** `::= DRAin <cap-id>`

Eject	
Preconditions	• service-level = FULL • lmu-status(lmu-id) = ONLINE • lsm-status(lsm-id) = ONLINE
Postconditions	• location(Volumes) = OUTSIDE
BNF	**eject-cmd** `::= EJect` {**VOLSER** | **SCRATCH**} **VOLSER** `::= {<volser> | <vol-range> | (<vol-list>)} [00 | <acs-id> | <cap-id>}` **SCRATCH** `::= SCRTCH [<acs-id> | <cap-id>] [SUBpool(<subpool-name>)] [VOLCNT(1) | VOLCNT(<vol-count>)]`

Enter	
Preconditions	• service-level = FULL • lmu-status(lmu-id) = ONLINE • lsm-status(lsm-id) = ONLINE
Postconditions	• cap-status(cap-id) = ENTERING • location(volsers) = INSIDE
BNF	**enter-cmd** `::= ENter [00 | <acs-id> | <cap-id>] [SCRatch]`

Journal	
Preconditions	• lmu-status(lmu-id) = ONLINE
Postconditions	
BNF	**journal-cmd** `::= Journal Full= {ABEND | Continue}`

Load	
Preconditions	• lmu-status(lmu-id) = ONLINE
Postconditions	
BNF	**load-cmd** `::= LOad {SLSLDQR | SLSMDQR}`

MNTD	
Preconditions	• service-level = FULL • lmu-status(lmu-id) = ONLINE

Postconditions	• One or more of the following:
	autoclean = changed value
	dismount = changed value
	float = changed value
	maxclean = changed value
	mountmsg = changed value
	scratch = changed value
	volwatch = changed value

BNF	**mntd-cmd** `::= MNTD` **MNTD-CHOICE [MNTD-CHOICE]** `[HOSTID(<host-id>)]` **MNTD-CHOICE** `::=` **{AUTOCLEAN | DISMOUNT | FLOAT | MAXCLEAN | MOUNTMSG | SCRATCH | VOLWATCH}** **AUTOCLEAN** `::= AUtocln({OFf | ON})` **DISMOUNT** `::= Dismount({Auto | Manual)}` **FLOAT** `::= Float({ON |OFf})` **MAXCLEAN** `::= MAXclean({100 |<count>})` **MOUNTMSG** `::= MOuntmsg({Roll | Noroll})` **SCRATCH** `::= Scratch({Manual | Auto})` **VOLWATCH** `::= VOLWatch({OFf | ON})`

Modify	
Preconditions	• service-level = FULL • lmu-status(lmu-id) = ONLINE
Postconditions	• lsm-status(lsm-id) = {ONLINE | OFFLINE}
BNF	**modify-cmd** `::= {MODify | F}` **LSM** `{ONline | OFFline [FORCE]}` **LSM** `::= {<lsm-id> | <lsm-range> | (<lsm-list>)}`

Monitor	
Preconditions	
Postconditions	• monitoring(cc or name) = ON
BNF	**monitor-cmd** `::= {MONITOR | MN} [PGMI] [,L= [<console> | <console-name>] ]`

Mount	
Preconditions	• service-level = FULL • lmu-status(lmu-id) = ONLINE • lsm-status(lsm-id) = ONLINE
Postconditions	location(volser) = drive-id
BNF	**mount-cmd** `::= Mount` **{VOLSER | SCRATCH}** **VOLSER** `::= <volser> <drive-id> [ {, | <host-id>} | [Readonly] ]` **SCRATCH** `::= {SCRTCH | PRIVAT} <drive-id> [<host-id>] [SUBpool(<subpool-name>)]`

Move	
Preconditions	• service-level = FULL • lmu-status(src-lmu-id) = ONLINE • lsm-status(dest-lsm-id) = ONLINE • CDS = ENABLED
Postconditions	• location(volsers) = changed
Intracommand Rule	• (Flsm = Tlsm) → Panel ≠ FPanel
BNF	**move-cmd** `::= MOVe` **{FROM-LSM | VOLSER} TO-LSM** **FROM-LSM** `::= Flsm(<lsm-id>) Panel(<pp>) {Row(<rr-list>) [Column(<cc>)] | Row(<rr>) [Column(<cc-list>)]` **VOLSER** `::= Volumn({<volser> | <vol-range> | <vol-list>})` **TO-LSM** `::= TLsm({<lsm-id> | <lsm-list>}) [TPanel(<pp>)]`

Recover	
Preconditions	• service-level = FULL
	• lmu-status(lmu-id) = ONLINE
Postconditions	
BNF	**recover-cmd** `::= RECover <host-id> [FORCE]`
Option	
Preconditions	• lmu-status(lmu-id) = ONLINE
Postconditions	• One or more of the following :
	entdup = Changed value
	output = Changed value
	viewtime = Changed value
BNF	**option-cmd** `::= OPTion` **OPTION-CHOICE** `[`**OPTION-CHOICE**`]`
	`[HOSTID(<host-id>)]`
	OPTION-CHOICE `::=` {**ENTDUP** | **OUTPUT** | **VIEWTIME** }
	ENTDUP `::= ENTdup({Auto | Manual})`
	OUTPUT `::= Output({Upper | Mixed})`
	VIEWTIME `::= Viewtime({10 | <viewcount>})`
Release	
Preconditions	• service-level = FULL
	• lmu-status(lmu-id) = ONLINE
	• lsm-status(lsm-id) = ONLINE
Postconditions	cap-status(cap-id) = DRAINED
BNF	**release-cmd** `::= RELease <cap-id>`
Retry	
Preconditions	lmu-status(lmu-id) = ONLINE
Postconditions	
BNF	**retry-cmd** `::= RETry {J3init | J}`
Scrparm	
Preconditions	• lmu-status(lmu-id) = ONLINE
Postconditions	• One or more of the following:
	initwarn = new value
	inittime = new value
	secwarn = new value
	sectime = new value
	baltol = new value
BNF	**scrparm-cmd** `::= SCRparm [`**INITWARN** | **INITTIME** |
	SECWARN | **SECTIME** | **BALTOL** `]`
	INITWARN `::= {0 | <initwarn>}`
	INITTIME `::= {4 | <inittime>}`
	SECWARN `::= {0 | <secwarn> }`
	SECTIME `::= {4 | <sectime>}`
	BALTOL `::= {1 | <baltol>}`
Senter	
Preconditions	• service-level = FULL
	• lmu-status(lmu-id) = ONLINE
	• lsm-status(lsm-id) = ONLINE
Postconditions	• location(volser) = INSIDE
BNF	**senter-cmd** `::= SENter <cap-id>`

Set	
Preconditions	• service-level = FULL • lmu-status(lmu-id) = ONLINE

Postconditions	• One or more of the following:
	autoclean = new value
	maxclean = new value
	dismount = new value
	entdup = new value
	float = new value
	mountmsg = new value
	output = new value
	scratch = new value
	vol-watch = new value

BNF	set-cmd ::= SET {CLEAN | DISMOUNT | ENTDUP | FLOAT | MOUNTMSG | OUTPUT | SCRATCH | VOLWATCH } CLEAN ::= CLean [Max [<count>] | ON | OFf] DISMOUNT ::= Dismount [Auto | Manual] ENTDUP ::= ENTdup [Manual | Auto] FLOAT ::= Float [ON | OFf] MOUNTMSG ::= MOuntmsg[Roll | Noroll] OUTPUT ::= Output [Upper | Mixed] SCRATCH ::= Scratch [Manual | Auto] VOLWATCH ::= VOLWatch [OFf | ON]

Srvlev	
Preconditions	
Postconditions	service-level = {BASE | FULL}
BNF	srvlev-cmd ::= SRVlev {BASE | FULL}

Stopmn	
Preconditions	
Postconditions	• monitoring(cc or name) = OFF
BNF	stopmn-cmd ::= {STOPMN | PM} [PGMI] [,L= [<console> | <console-name>]]

Switch	
Preconditions	• service-level = FULL • lmu-status(lmu-id) = ONLINE • Dual LMU Configuration
Postconditions	• lsm-status(Standby LSM) = ONLINE • lsm-status(Master LSM) = Standby
BNF	switch-cmd ::= SWitch [Acs <acs-id>]

Trace	
Preconditions	
Postconditions	• trace(comp-name) = {TRACE | NOTRACE }
BNF	trace-cmd ::= TRace [[OFF] {<comp-name> | <comp-list>}]

Uexit	
Preconditions	• lmu-status(lmu-id) = ONLINE
Postconditions	• uexit-status(uexit-number or uexit-name) = {Enable | Disable}

BNF	uexit-cmd ::= UEXIT {UEXIT-LOAD | UEXIT-NUMBER }
	UEXIT-LOAD ::= <uexit-number> {LOAD | Enable | Disable}
	LOAD ::= Load [=SLSUX<uexit-number> | =<uexit-name>] [,Enable | ,Disable]
	UEXIT-NUMBER ::= {<uexit-number> | (<uexit-range>) | (<uexit-list>) } Query

Vary	
Preconditions	• service-level = FULL
Postconditions	• lmu-status(lmu-id) = {ONLINE | OFFLINE}
BNF	vary-cmd `::= Vary {<lmu-id> | <lmu-range> | (<lmu-list>)} {ONline | OFFline [FORCE] }`

Warn	
Preconditions	• service-level = FULL • lmu-status(lmu-id) = ONLINE

Postconditions	• One of the following:
	acs-scr-threshold = changed
	acs-subpool-threshold = changed
	lsm-scr-threshold = changed
	lsm-subpool-threshold = changed

BNF	`warn-cmd ::= Warn SCRatch {<acs-id> | <lsm-id>} [SUBpool(<subpool-name>)] THRESHold(<threshold>)`

View	
Preconditions	• service-level = FULL
Postconditions	
BNF	view-cmd `::= VIew` {CAP | CELL | DRIVE | PLAYGND | PTP } `[Time(<time>)]` CAP `::= CAp [Lsm(000) | Lsm(<lsm-id>)] [Row(00) | Row(<rr>)] [Column(00) | Column(<cc>)]` CELL `::= CEll [Lsm(000) | Lsm(<lsm-id>)] [Panel(00) | Panel(<pp>)] [Row(00) | Row(<rr>)] [Column(00) | Column(<cc>)]` DRIVE `::= DRive Address(<drive-id>) [Host(<host-id>)]` PLAYGND `::= PLaygrnd [Lsm(000) | Lsm(<lsm-id>)] [Column(00) | Column(<cc>)]` PTP `::= PTp [Lsm(000) | Lsm(<lsm-id>)][Xlsm(<ptp-id>)] [Column(00) | Column(<cc>)]`

A.3 Object Element Glossary

The StorageTek HSC 1.2 command language has 45 parameters. In Tables A.2-A.11 all parameters are defined by object element type, default values, aliases, initial value, and a representation.

Table A.2: Parameter Dictionary #1

Parameter Name		
acs-id	Full Name	Automated Cartridge System (ACS) Identifier
	Definition	Names an Instance of an ACS
	Type	parameter attribute
	Values	range = [00..FF]
	Object	ACS
	Representation	Range of values
	Number of Values	1 to 16
acs-scr-threshold	Definition	If the number of scratch cartridges falls below this threshold on the ACS then a warning will be issued.
	Type	parameter mode
	Values	range = [0..9999]
	Object	ACS
	Representation	Range
acs-subpool-threshold	Definition	If the number of scratch cartridges falls below this threshold in the subpool on the ACS then a warning will be issued.
	Type	parameter mode
	Values	range = [0..9999] plus the subpool-name
	Object	ACS
	Representation	Range and subpool-name pair
autoclean	Definition	Set HSC automatic tape transport cleaning
	Type	parameter state
	Values	ON | OFF
	Initial Value	ON
	Object	HSC
	Representation	Enumeration
baltol	Definition	Set scratch redistribution level
	Type	parameter mode
	Values	range = [1..9] initial value = 1
	Object	HSC
	Representation	Range
cap-id	Full Name	Cartridge Access Port (CAP) Identifier
	Definition	Names an Instance of a CAP
	Type	parameter attribute
	Values	aal, where aa = acs-id and l = lsm number
	Object	CAP
	Representation	Inherited range
	Number of Values	1 to 16
cap-status	Definition	Status of the Cartridge Access Port (CAP)
	Type	non-parameter state
	Values	DRAINED | ENTERING | EJECTING
	Object	CAP

Table A.3: Parameter Dictionary #2

Parameter Name		
cc (Column)		
	Full Name	Column Number
	Definition	Names an Instance of a Column in a Panel
	Type	parameter attribute
	Values	range = [00,01,02,..23] for outer panels
		range = [00..19] for inner panels
	Object	Column
	Representation	Range of values
	Number of Values	1 to 23 for each panel
cc (Console)		
	Full Name	Console Identifier
	Definition	Names an Instance of a Console
	Type	parameter attribute
	Internal Name	console-id
	Values	range = [00..FF]
	Object	Console
	Representation	List
	Expect Number of Values	1 or 2
command-name		
	Full Name	HSC Command Name
	Definition	Name of an HSC Command
	Type	parameter attribute
	Values	List of Names = [CAPPref, CDS, ENter, etc]
	Object	Documentation
	Representation	Enumeration
	Expect Number of Values	30
comp-name		
	Definition	Name of an HSC Component for which tracing is to be enabled or disabled.
	Type	parameter mode
	Values	ALlocati|AScomm|CAp|COnfigur|Database| HComm Initiali|JES3Aloc|JES3Dira|JES3Msgs| JES3Sep|Job Lmu|Mount|Operator|Recovery| Utilitie|Volume|Wto
	Object	HSC
	Representation	Binary State Vector
deferred		
	Definition	Set deferred mount processing
	Type	parameter mode
	Values	ON | OFF | JES3
	Initial Value	OFF
	Object	HSC
	Representation	Enumeration

Table A.4: Parameter Dictionary #3

Parameter Name		
dismount	Definition	Specifies whether volumes are to be automatically deleted from the control data set when a dismount is requested in a manual mode LSM for a volume that was mounted by the robot before the LSM was modified
	Type	parameter mode
	Values	AUTO | MANUAL
	Initial Value	AUTO
	Object	HSC
	Representation	Enumeration
drive	Full Name	Device Address of the tape transport
	Alias	devaddr (MOUNT, DISMOUNT) dev-id (CLEAN) xxx (VIEW)
	Definition	Names an Instance of a tape drive
	Type	parameter attribute
	Internal Name	drive
	Values	000...FFF
	Object	Tape Transport
	Representation	File or Enumeration, Sets of Ranges
	Number of Values	Max of 256
drive-status	Definition	Status of tape transport (tape drive)
	Type	non-parameter state
	Values	BUSY | AVAILABLE
	Object	Tape Transport
	Representation	Enumeration
dsn	Full Name	Control Data Set Name
	Definition	Names an Instance of a Control Data Set
	Type	parameter attribute
	Values	Alphanumeric
	Object	CDS
	Representation	File
	Number of Values	???
entdup	Definition	Specifies whether the HSC prompts the operator when an enter operation finds duplicate volser in the control data set, but cannot locate the cartridge in the ACS. The options specify automatic or manual deletion of the duplicate volume.
	Type	parameter mode
	Values	AUTO | MANUAL
	Initial Value	AUTO
	Object	HSC
	Representation	Enumeration

Table A.5: Parameter Dictionary #4

Parameter Name		
float	Definition	Allows the HSC to select a new home cell location when it dismounts a volume that required a pass-thru when it was mounted.
	Type	parameter mode
	Values	ON | OFF
	Initial Value	ON
	Object	HSC
	Representation	Enumeration
full-journal	Definition	Describes system action to take when a system journal becomes full
	Type	parameter mode
	Values	ABEND | CONTINUE
	Object	HSC
	Representation	Enumeration
gdg-sep	Definition	Unit affinity separation for GDG chains
	Type	parameter state
	Values	SEP | NOSEP
	Initial Value	NOSEP
	Object	HSC
	Representation	Enumeration
host-id	Full Name	Host Identifier
	Definition	Names an Instance of a Host
	Type	parameter attribute
	Values	nnnn, Example: MVSE, MVS1, MVSF, HST1, HST3
	Object	HSC
	Representation	File or Enumeration
	Number of Values	1 to 16 (possibly more)
inittime	Definition	Time interval in minutes between checks of the number of scratch cartridges
	Type	parameter mode
	Values	range = [1..99] initial value = 4
	Object	HSC
	Representation	Range
initwarn	Definition	If the number of scratch cartridges in an ACS drops below this threshold a warning message is issued.
	Type	parameter mode
	Values	range = [0..9999] initial value = 0
	Object	HSC
	Representation	Range

Table A.6: Parameter Dictionary #5

Parameter Name		
journal-full	Definition	A dynamic event that results when the system journals become full
	Type	nonparameter event
	Values	NOT-FULL | FULL
	Object	HSC
	Representation	Enumeration
lmu-status	Definition	Status of the Library Management Unit (LMU) Station
	Type	parameter state
	Values	UP | DOWN
	Object	LMU
	Representation	Enumeration
lsm-id	Full Name	Library Storage Module (LSM) Identifier
	Definition	Names an Instance of an LSM within an ACS
	Type	parameter attribute
	Values	000..FFF
	Object	LSM
	Representation	Range of values
lsm-scr-threshold	Definition	If the number of scratch cartridges falls below this threshold on the LSM then a warning will be issued.
	Type	parameter mode
	Values	range = [0..9999]
	Object	LSM
	Representation	Range
lsm-status	Definition	Status of the Library Storage Module (LSM) Station
	Type	parameter state
	Values	ONLINE | OFFLINE
	Object	LSM
	Representation	Enumeration
lsm-subpool-threshold	Definition	If the number of scratch cartridges falls below this threshold in the subpool on the. LSM then a warning will be issued.
	Type	parameter mode
	Values	range = [0..9999] plus the subpool-name
	Object	LSM
	Representation	Range and subpool-name pair
maxclean	Definition	Number of times a cleaning cartridge is used before ejecting
	Type	parameter mode
	Values	10..500
	Initial Value	100
	Object	HSC
	Representation	Range

Table A.7: Parameter Dictionary #6

Parameter Name		
mount-msg	Definition	Allows messages to scroll off operator's screen before mount requests are satisfied.
	Type	parameter mode
	Values	ROLL | NOROLL
	Initial Value	ROLL
	Object	HSC
	Representation	Enumeration
msg-id	Full Name	Message Identifier
	Definition	Identifies the four-digit portion of the message identifier
	Type	parameter attribute
	Values	nnnn. Leading zeros are not required
	Object	Documentation
	Representation	File
	Number of Values	
name (Console)	Full Name	Console Name
	Definition	Specifies the name of the console for MVS/SP 4.1.0 or higher
	Internal Name	console-name
	Type	parameter attribute
	Values	alphanumeric
	Object	Console
	Representation	File
	Number of Values	1
name (HSC)	Full Name	User Exit Module Name
	Definition	Specifies the name of the of the user defined exit load module
	Internal Name	uexit-name
	Type	parameter attribute
	Values	File
	Object	HSC
	Representation	File
	Number of Values	
nn	Full Name	User Exit Number
	Definition	Specifies the exit number for a user defined exit load module
	Internal Name	uexit-number
	Type	parameter attribute
	Values	1-10
	Object	HSC
	Representation	Range
	Number of Values	1-10

Table A.8: Parameter Dictionary #7

Parameter Name		
output	Definition	Output messages to operator's console in uppercase or upper/lower case
	Type	parameter mode
	Values	UPPER | MIXED
	Initial Value	UPPER
	Object	HSC
	Representation	Enumeration
pp	Full Name	Panel Number
	Definition	Names an Instance of a Panel in an LSM
	Type	parameter attribute

	Values	range = [00,01,02,..11] for outer panels
		range = [12..19] for inner panels

	Object	Panel
	Representation	Range of Values or Enumeration
	Number of Values	1 to 20 for each LSM
prefvlu	Definition	Preference value for the Cartridge Access Port (CAP)
	Type	parameter state
	Values	range = [0..9] 9 is the highest preference
	Initial Value	0
	Object	CAP
	Representation	Range
ptp-id	Full Name	Pass Through Port Identifier
	Definition	Names a Pass Through Port in an LSM
	Type	parameter attribute
	Values	l, where l = lsm number
	Object	Pass Through Port
	Representation	Range of Values or Enumeration
	Number of Values	1 to 8 for each LSM
rr	Full Name	Row Number
	Definition	Names an Instance of a Row in a Panel
	Type	parameter attribute

	Values	range = [00..14] for outer panels
		range = [00..05,08..14] for inner panels

	Object	Row
	Representation	Range of Values or Enumeration
	Number of Values	1 to 15 for each panel
scratch	Definition	Determines how a scratch volume is selected to satisfy a scratch mount request for a manual mode LSM
	Type	parameter mode
	Values	AUTO | MANUAL
	Initial Value	MANUAL
	Object	HSC
	Representation	Enumeration

Table A.9: Parameter Dictionary #8

Parameter Name		
separation	Definition	Set unit affinity separation
	Type	parameter state
	Values	SEP | NOSEP
	Initial Value	NOSEP
	Object	HSC
	Representation	Enumeration
sectime	Definition	Time period in minutes between checks of the scratch pool after the number of cartridges in the ACS drops below the initial warning level
	Type	parameter mode
	Values	range = [1..99] initial value = 4
	Object	HSC
	Representation	Range
secwarn	Definition	If the number of scratch cartridges in the ACS drops below the initial value, the secondary warning value is used to trigger additional messages.
	Type	parameter mode
	Values	range = [0..99] initial value = 0
	Object	HSC
	Representation	Range
service-level	Definition	Specify the service level for HSC operations
	Type	parameter state
	Values	BASE | FULL
	Object	HSC
	Representation	Enumeration
specvol	Definition	Transports available when no non-library drives exist
	Type	parameter state
	Values	YES | NO
	Object	HSC
	Representation	Enumeration
station	Full Name	Library Management Unit (LMU) station
	Definition	Names an Instance of an LMU Station
	Alias	dev-id (VARY)
	Type	parameter attribute
	Values	alphanumeric
	Object	LMU
	Representation	Enumeration, Set of Ranges
	Number of Values	1 to 16 per LMU

Table A.10: Parameter Dictionary #9

Parameter Name		
subpool-name	Full Name	Subpool Name
	Definition	Identifies a subpool
	Type	parameter attribute
	Values	alphanumeric
	Object	Scratch Pool
	Representation	File
	Number of Values	256
subpool-threshold	Definition	If the number of scratch cartridges falls below this threshold in the subpool then a warning will be issued.
	Type	parameter mode
	Values	range = [0..9999] plus the subpool-name
	Object	Scratch Pool
	Representation	Range and subpool-name pair
viewtime	Definition	Time in seconds to focus the camera on a specified element
	Type	parameter mode
	Values	range = [5..120] initial value = 10
	Object	HSC
	Representation	Range
volcnt	Full Name	Volume Count
	Definition	Total number of volumes to eject
	Type	parameter attribute
	Values	range = [1..100]
	Object	Cartridge
	Representation	Range of values
	Number of Values	Pick from the range
volser	Full Name	Volume Serial Number
	Definition	Names an Instance of a Tape Cartridge
	Type	parameter attribute
	Values	1 to 6 characters in [A-Z0-9#] Trailing blanks to fill out to 6 characters
	Object	Cartridge
	Representation	File
	Number of Values	A lot (hundreds...)

Table A.11: Parameter Dictionary #10

Parameter Name		
vol-watch	Definition	Set HSC messages when mount for library volume requested on a non-library device
	Type	parameter mode
	Values	ON | OFF
	Initial Value	OFF
	Object	HSC
	Representation	Enumeration
zeroscr	Definition	Restricts device selection for requested scratch mounts
	Type	parameter state
	Values	ON | OFF
	Initial Value	OFF
	Object	HSC
	Representation	Enumeration

A.4 Script Defintion

A.4.1 Script Classes

Scripting classes can be partitioned by function, object, and object element. Functional partitioning creates scripting classes that include commands that perform similar actions. For example, in the **StorageTek** domain, the *set-up* class includes all commands that perform system set up functions; the *action* class includes commands that manipulate exercise the robot tape library; the *mode* class that sets system operating modes; and and the *any* class represents the universal set that contains all commands from the command language.

A.4.2 Script Rules

Scripting rules and script parameter binding capture dynamic system behavior. The domain model for HSC Release 1.2 used four script rules. Table A.13 list them and their parameter bindings.

Table A.12: Script Classes for the **StorageTek** HSC Domain

Script Class	Commands						
Any	Alloc	Commpath	Eject	Mntd	Move	Retry	Srvlev
	Cappref	Dismount	Enter	Modify	Option	Scrparm	Switch
	Cds	Display	Journal	Monitor	Recover	Senter	Trace
	Clean	Drain	Load	Mount	Release	Set	Uexit
Mode	Cappref	Clean	Mntd	Option	Set	Trace	Warn
	Cds	Journal	Monitor	Scrparm	Stopmn	Uexit	
Set-Up	Alloc	Journal	Option	Srvlev	Trace	Cappref	Mntd
	Scrparm	Stopmn	Uexit	Commpath	Modify	Set	Switch
	Vary						
Action	Alloc	Display	Enter	Move	Retry	Commpath	Drain
	Load	Recover	Senter	Dismount	Eject	Mount	Release
	View						

Table A.13: Script Rules with Paramenter Binding

Command Name	Script Rule
Mount	`MOUNT [tape-id*] [drive-id*] <5/any> DISMOUNT [tape-id] [drive-id]`
Dismount	`MOUNT [tape-id*] [drive-id*] <5/any> DISMOUNT [tape-id] [drive-id]`
Enter	`ENTER [cap-id*] <5/any> DRAIN [cap-id]`
Drain	`ENTER [cap-id*] <5/any> DRAIN [cap-id]`

Appendix B

SLEUTH TEST GENERATION TIME DATA

Table B.1 lists all data collected during the *Sleuth* timing study. Three test subdomains were used in the experiments: Full Domain, No Script Rules, No Semantic Rules. For each test subdomain, we measured test generation time for test cases lengths of 50, 100, 250, and 500 commands. Experiments with the Full Domain generate more commands than requested because of scripting rule expansion. Table B.2 shows the requested versus actual test case lengths from the Full Domain.

Table B.1: *Sleuth* Timing Study - Time in Seconds

Subdomain	Requested Test Case Length			
	50	100	250	500
Full Domain (Time in Seconds)	99	167	475	837
	125	183	503	1220
	90	225	618	908
	83	160	440	988
	124	169	396	926
	85	189	516	1082
	138	208	513	825
	90	199	525	853
	113	195	535	1038
	146	260	395	993
Average	80	109	491	967

Subdomain	Requested Test Case Length			
	50	100	250	500
No Script Rules (Time in Seconds)	32	95	221	424
	48	75	213	422
	36	98	226	441
	39	79	252	391
	39	89	216	402
	41	92	219	405
	48	102	215	409
	35	76	209	419
	42	95	202	439
	44	89	225	454
Average	40	89	220	421

Subdomain	Requested Test Case Length			
	50	100	250	500
No Semantic Rules (Time in Seconds)	48	83	223	369
	38	82	210	372
	47	81	217	371
	48	80	231	384
	48	84	227	350
	41	80	271	378
	46	84	255	411
	43	82	199	421
	48	86	255	405
	46	88	227	375
Average	45	83	232	384

Table B.2: *Sleuth* Timing Study - Requested vs Actual Test Case Length (Full Domain)

Number Requested	Actual	Number Requested	Actual
50	88	100	179
	127		186
	86		224
	87		252
	127		181
	100		199
	137		219
	85		205
	106		204
	152		197
Average	110	Average	205

Number Requested	Actual	Number Requested	Actual
250	530	500	914
	498		1253
	605		933
	467		991
	432		948
	537		1127
	523		965
	548		979
	563		1099
	446		1011
Average	515	Average	1022

Appendix C

NEURAL NETWORK TRAINING DATA

The test case attributes used for the 21 input nodes and the 4 output notes are listed in Table C.1. The first input value is the length of the test case, the next ten inputs represent the relative frequency of each command in the test , and the last ten input vector values count the number of unique values for each parameter. The output vector classifies fault severity levels ranging from most severe (Severity 1) to least severe (Severity 3). Severity 4 indicates the test case is did not identify any faults. All 180 patterns used to train and evaluate the neural network are listed after the tables.

Table C.1: Input Vector and Output Vector Description

Vector Index	Description
Input	
1.	Test Case Length
2.	CDS Frequency
3.	DISMOUNT Frequency
4.	DISPLAY Frequency
5.	DRAIN Frequency
6.	EJECT Frequency
7.	ENTER Frequency
8.	MODIFY Frequency
9.	MOUNT Frequency
10.	MOVE Frequency
11.	SRVLEV Frequency
12.	acs Frequency
13.	cap Frequency
14.	cc Frequency
15.	drive Frequency
16.	dsn Frequency
17.	host Frequency
18.	lsm Frequency
19.	pp Frequency
20.	rr Frequency
21.	volser Frequency

Vector Index	Description
Output	
22.	Severity 1 Indicator
23.	Severity 2 Indicator
24.	Severity 3 Indicator
25.	Severity 4 Indicator

13	1	1	3	1	0	1	2	1	2	1	0	2	2	2	0	0	2	3	2	3	0	0	0	1
44	1	4	12	3	2	3	4	4	8	3	0	3	4	4	1	0	2	6	4	4	0	0	1	0
30	1	3	7	4	2	4	1	3	3	2	0	3	3	5	1	0	2	4	3	5	0	0	0	1
14	0	1	5	1	1	1	2	1	1	1	0	2	0	1	0	0	2	0	0	3	0	0	0	1
15	2	2	1	1	1	1	2	2	3	0	0	2	1	2	1	0	1	2	1	4	0	0	0	1
19	1	2	5	1	0	1	2	2	3	2	0	1	3	2	1	0	2	5	3	3	0	0	0	1
13	1	0	4	1	0	1	0	0	5	1	0	1	3	0	1	0	2	4	3	1	0	0	0	1
22	1	2	5	2	0	2	3	2	4	1	0	2	3	4	1	0	2	5	4	2	0	0	0	1
25	1	2	5	2	1	2	4	2	4	2	0	2	3	3	0	0	2	5	4	5	0	0	0	1
21	4	1	5	2	0	2	1	1	3	2	0	2	3	2	2	0	2	4	3	2	0	0	0	1
15	0	1	3	2	0	2	1	1	0	5	0	1	0	2	0	0	2	0	0	2	0	0	0	1
46	5	4	7	4	3	4	6	4	8	1	0	3	6	5	1	0	2	7	5	5	1	0	1	0
15	0	1	3	2	0	2	1	1	0	5	0	1	0	2	0	0	2	0	0	2	0	0	0	1
12	0	1	2	1	1	1	2	1	2	1	0	2	2	1	0	0	2	2	2	3	0	0	1	0
51	2	6	6	4	1	4	7	6	8	7	0	2	7	6	2	0	2	6	5	5	0	0	1	0
32	1	6	7	1	2	1	1	6	5	2	0	3	4	4	0	0	2	5	4	6	1	0	1	0
18	0	1	4	2	1	2	4	1	2	1	0	3	2	1	0	0	2	2	2	4	0	1	0	0
13	2	1	4	1	0	1	2	1	1	0	0	1	0	2	1	0	2	0	0	4	0	0	1	0
26	1	2	7	3	1	3	2	2	4	1	0	2	3	3	1	0	2	5	3	4	0	0	0	1
15	2	0	2	3	0	3	2	0	2	1	0	3	1	0	1	0	2	2	1	1	0	0	0	1
10	0	0	4	0	0	0	0	0	4	2	0	0	2	0	0	0	2	2	2	2	0	0	0	1
14	0	1	4	1	1	1	1	1	2	2	0	2	2	1	0	0	2	3	1	2	0	0	0	1
14	2	1	3	1	0	0	1	1	3	2	0	1	2	1	1	0	2	2	2	3	0	0	0	1
36	0	5	9	3	1	2	2	5	6	3	0	2	4	8	0	0	2	7	5	5	1	0	1	0
29	0	3	6	2	5	3	3	3	1	3	0	3	1	5	0	0	2	2	1	5	0	1	1	0
45	2	6	15	2	0	2	3	6	7	2	0	2	5	7	1	0	2	6	5	4	1	0	1	0
12	0	1	2	1	2	1	0	1	4	0	0	3	3	2	0	0	2	3	2	3	0	0	0	1
19	0	1	4	1	0	2	4	1	3	3	0	2	1	2	0	0	2	1	1	2	0	1	1	0
54	3	8	9	0	1	2	8	8	9	6	0	2	6	6	1	0	2	8	6	5	1	1	1	0
10	1	0	2	0	0	0	3	0	4	0	0	0	3	0	1	0	2	5	3	0	0	0	0	1
48	5	8	7	2	3	2	4	8	5	4	0	2	3	6	3	0	2	4	3	5	1	0	1	0
28	2	4	6	0	1	4	4	4	0	3	0	3	0	5	0	0	2	0	0	4	0	1	0	0
17	1	1	5	0	2	1	3	1	2	1	0	2	0	1	0	0	2	0	0	4	0	1	1	0
35	6	4	8	0	3	3	2	4	2	3	0	3	1	4	4	0	2	2	1	5	1	1	1	0
46	0	4	8	3	4	1	4	4	9	9	0	3	7	5	0	0	2	7	5	6	0	0	1	0
38	3	2	5	4	4	0	5	2	11	2	0	3	6	4	1	0	2	7	7	5	0	0	1	0
14	2	1	2	0	1	0	3	1	2	2	0	1	1	1	2	0	2	2	1	1	0	0	0	1
28	0	3	6	2	1	1	5	3	4	3	0	2	4	3	0	0	2	5	3	3	0	0	0	1
42	2	5	9	2	1	3	1	5	12	2	0	3	9	6	1	0	2	6	6	5	0	1	1	0
19	0	1	5	1	0	1	3	1	6	1	0	2	4	2	0	0	2	5	4	4	0	1	0	0
28	3	1	8	3	0	3	4	1	3	2	0	2	2	2	2	0	2	2	2	4	0	1	1	0
11	0	1	3	0	0	0	0	1	4	2	0	0	3	1	0	0	2	4	3	1	0	0	0	1
21	2	1	3	2	2	2	0	1	6	2	0	2	3	1	2	0	2	5	4	5	0	0	0	1
6	0	0	0	0	0	0	2	0	1	3	0	0	1	0	0	0	2	2	1	0	0	0	1	0
11	0	1	4	0	2	0	0	1	2	1	0	2	2	1	0	0	2	4	2	0	0	0	0	1
31	2	5	7	2	1	2	1	5	3	3	0	3	2	6	0	0	2	3	1	5	1	0	1	0
8	1	0	0	1	1	1	0	0	1	3	0	1	1	0	1	0	1	1	1	1	0	0	1	0
18	1	1	2	2	1	2	1	1	4	3	0	2	2	1	1	0	2	2	2	3	0	0	0	1
7	0	0	1	0	0	0	1	0	5	0	0	0	4	0	0	0	2	4	4	1	0	0	0	1
34	1	3	6	3	3	3	1	3	7	4	0	3	6	4	1	0	2	8	6	4	0	1	1	0
24	0	4	4	0	0	0	0	4	10	2	0	0	5	4	0	0	2	7	5	4	0	1	0	0
44	2	2	12	5	2	5	5	2	8	1	0	3	6	4	1	0	2	5	5	5	0	0	1	0
27	1	1	1	3	1	3	7	1	6	3	0	2	4	1	1	0	2	5	3	4	0	1	1	0
25	2	3	8	2	0	2	2	3	3	0	0	2	1	3	0	0	2	2	1	5	0	0	0	1
32	1	1	5	4	2	4	0	1	10	4	0	3	5	1	1	0	2	8	4	3	0	1	1	0
82	4	14	17	4	4	4	5	14	11	5	0	3	3	8	2	0	2	6	3	6	1	0	1	0

16	0	2	3	1	1	1	3	2	2	1	0	2	2	3	0	0	2	3	2	3	0	0	1	0
43	3	2	5	5	3	5	2	2	10	6	0	3	5	3	0	0	2	6	6	6	0	0	1	0
22	1	4	7	1	1	1	2	4	1	0	0	2	0	4	0	0	2	0	0	4	0	0	0	1
19	1	3	2	1	1	1	1	3	4	2	0	3	4	4	0	0	2	5	3	5	0	1	0	0
13	1	0	4	1	0	1	2	0	2	2	0	1	2	0	1	0	2	4	2	1	0	0	0	1
31	0	2	8	4	0	4	4	2	5	2	0	3	4	2	0	0	2	5	3	4	0	1	1	0
21	1	3	8	1	1	1	0	0	5	1	0	2	2	3	1	0	2	4	2	4	0	1	0	0
23	2	0	6	1	1	1	4	2	5	1	0	3	4	2	0	0	2	3	3	3	0	1	1	0
23	0	1	1	2	1	2	5	3	8	0	0	3	5	3	0	0	2	6	6	4	0	0	1	0
39	1	0	12	4	3	4	0	4	8	3	0	3	4	3	0	0	2	4	3	5	0	0	0	1
27	2	1	9	2	1	2	6	0	2	2	0	2	2	1	2	0	2	3	1	2	0	1	1	0
15	2	0	4	0	1	0	0	3	3	2	0	1	3	3	1	0	2	5	3	2	0	0	0	1
19	1	0	6	3	1	3	1	1	2	1	0	2	2	1	0	0	2	3	2	3	0	0	0	1
39	3	2	10	5	2	5	1	1	8	2	0	3	5	1	3	0	2	4	5	4	0	1	0	0
10	1	1	3	0	0	0	1	1	3	0	0	0	2	2	1	0	2	3	1	3	0	0	0	1
24	2	1	7	2	2	2	1	2	3	2	0	2	1	2	1	0	2	1	1	4	0	0	0	1
17	1	2	5	1	0	1	2	0	3	2	0	1	3	1	1	0	2	3	3	2	0	0	1	0
26	1	1	8	2	2	2	0	2	7	1	0	3	4	2	1	0	2	5	3	2	0	0	0	1
25	0	1	7	0	0	0	6	2	7	2	0	0	6	2	0	0	2	5	6	4	0	0	1	0
19	0	0	2	2	2	2	4	1	2	4	0	3	2	1	0	0	2	4	1	1	0	0	1	0
49	2	1	8	7	4	7	2	3	10	5	0	3	7	3	2	0	2	7	6	4	0	1	1	0
71	3	4	15	8	6	8	6	4	12	5	0	3	6	4	1	0	2	6	4	5	0	1	1	0
45	2	2	10	5	0	5	4	5	11	1	0	2	6	4	1	0	2	5	6	5	0	1	0	0
30	2	1	6	3	3	3	5	3	3	1	0	2	2	3	0	0	2	2	2	5	0	1	1	0
10	0	0	3	0	1	0	1	2	0	3	0	1	0	2	0	0	2	0	0	4	0	0	0	1
35	2	3	11	1	2	2	2	4	5	3	0	3	2	4	2	0	2	3	2	6	1	1	0	0
35	1	1	6	5	3	2	1	1	12	3	0	3	4	1	0	0	2	8	5	6	0	1	1	0
35	4	0	10	1	3	4	2	1	7	3	0	3	3	1	2	0	2	5	3	5	0	1	1	0
45	2	6	12	1	3	4	6	1	8	2	0	3	5	4	1	0	2	10	5	7	0	1	1	0
45	4	2	11	0	3	1	6	3	10	5	0	2	4	3	2	0	2	7	4	7	0	1	1	0
45	5	2	10	0	3	0	11	0	9	5	0	2	7	2	2	0	2	10	5	5	0	1	1	0
55	5	4	9	4	2	5	7	1	14	4	0	3	9	4	3	0	2	14	7	6	0	1	1	0
55	4	5	13	3	3	2	7	3	13	2	0	3	9	5	4	0	2	8	7	6	0	1	1	0
55	2	3	9	2	5	5	6	5	17	1	0	3	8	5	1	0	2	13	6	7	0	1	1	0
30	2	1	8	3	1	1	4	1	6	3	0	2	2	2	0	0	2	6	3	5	0	0	0	1
10	0	0	5	0	0	1	1	0	3	0	0	1	3	0	0	0	1	4	3	0	0	0	1	0
30	2	1	13	2	2	2	2	2	2	2	0	2	2	2	2	0	2	4	2	5	0	0	0	1
10	1	2	4	0	0	0	1	0	1	1	0	0	1	2	1	0	2	2	1	3	0	0	0	1
20	1	1	5	1	0	2	0	0	7	3	0	1	3	1	1	0	2	6	4	5	0	1	0	0
20	0	0	3	1	0	1	4	1	7	3	0	2	5	1	0	0	2	5	5	3	0	0	1	0
20	0	2	5	0	1	0	2	1	6	3	0	1	4	2	0	0	2	6	3	5	0	1	1	0
25	1	4	5	3	2	0	3	0	5	2	0	3	4	3	0	0	2	9	4	4	0	1	1	0
25	2	0	6	2	0	0	8	1	5	1	0	1	3	1	2	0	2	4	3	3	0	0	1	0
25	1	4	6	0	3	2	2	2	3	2	0	2	3	4	1	0	2	4	3	5	0	1	1	0
10	1	0	4	0	1	0	2	0	1	1	0	1	0	0	0	0	2	0	0	1	0	0	0	1
25	2	3	2	3	0	0	2	2	7	4	0	3	7	5	2	0	2	8	5	2	0	1	0	0
25	1	2	5	1	1	2	6	2	4	1	0	2	3	4	0	0	2	6	3	4	0	1	0	0
25	2	1	6	1	1	3	3	0	4	4	0	2	3	1	2	0	2	5	4	4	0	1	0	0
30	2	1	6	0	1	4	4	1	7	4	0	3	6	1	2	0	2	6	6	3	0	1	1	0
30	1	2	5	1	1	2	2	3	10	3	0	2	7	4	0	0	2	7	6	4	0	1	1	0
30	0	3	10	1	0	3	4	1	6	2	0	3	4	3	0	0	2	5	3	2	0	1	1	0
40	8	5	10	0	3	1	6	1	4	2	0	3	4	4	3	0	2	4	4	5	0	1	1	0
40	3	5	6	2	2	1	5	1	14	1	0	2	9	4	1	0	2	7	7	6	0	1	1	0
40	2	1	9	3	2	3	3	4	9	4	0	3	5	5	2	0	2	7	5	6	0	1	1	0
50	4	2	9	4	0	2	9	2	12	6	0	3	8	4	2	0	2	8	6	4	0	0	1	0
10	1	0	1	0	1	2	1	1	1	2	0	2	0	1	0	0	1	0	0	1	0	0	1	0
50	1	1	12	6	2	2	8	5	5	8	0	3	5	5	0	0	2	7	5	4	1	1	0	0

10	0	1	3	2	0	0	2	1	1	0	0	2	1	2	0	0	2	2	1	2	0	0	1	0
15	0	0	6	0	0	2	0	1	4	2	0	2	3	1	0	0	2	4	3	2	0	1	0	0
15	2	1	3	1	1	2	2	1	1	1	0	3	1	2	1	0	2	1	1	3	0	1	1	0
15	1	1	4	1	0	1	1	3	2	1	0	2	1	3	1	0	2	2	1	3	0	0	0	1
20	1	1	5	1	1	3	0	2	5	1	0	3	3	3	1	0	2	4	2	3	0	1	0	0
8	0	2	1	0	2	0	0	0	3	0	0	1	2	2	0	0	2	3	2	4	0	0	0	1
20	1	2	1	0	0	2	6	0	7	1	0	2	5	2	1	0	2	6	5	2	0	1	0	0
10	0	0	3	0	2	0	1	0	2	2	0	2	1	0	0	0	2	2	1	3	0	0	1	0
16	2	1	0	1	1	1	3	1	6	0	0	2	4	1	1	0	2	4	5	3	0	0	1	0
67	6	9	14	3	2	3	6	9	9	6	0	3	6	7	3	0	2	9	6	6	1	1	1	0
55	0	9	8	4	5	4	7	9	7	2	0	3	5	6	0	0	2	6	4	6	1	0	1	0
32	1	5	5	2	1	2	1	5	7	3	0	2	6	5	0	0	2	5	5	6	0	0	1	0
8	0	0	3	0	0	0	2	0	2	1	0	0	2	0	0	0	2	4	2	1	0	0	1	0
22	1	1	3	2	2	2	2	1	4	4	0	2	3	2	1	0	2	5	3	4	0	0	1	0
18	0	1	2	1	1	1	5	1	4	2	0	2	1	1	0	0	2	4	2	3	0	0	1	0
32	1	3	7	3	0	3	2	3	8	2	0	2	4	4	1	0	2	5	4	4	0	0	1	0
52	3	5	11	2	5	2	4	5	8	7	0	3	6	4	3	0	2	6	4	6	0	0	1	0
30	0	0	8	6	1	6	2	0	3	4	0	3	3	0	0	0	2	5	3	1	0	0	1	0
16	0	1	4	1	0	1	2	0	5	2	0	1	3	1	0	0	2	4	3	3	0	0	0	1
29	0	2	5	4	3	4	3	0	6	2	0	3	4	2	0	0	2	5	5	5	0	1	0	0
10	0	1	1	0	1	0	4	0	3	0	0	1	2	1	0	0	2	1	1	3	0	0	1	0
24	2	0	4	3	1	3	4	1	5	1	0	3	4	1	1	0	2	4	4	2	0	1	0	0
18	0	3	2	2	3	2	0	1	3	2	0	3	3	3	0	0	2	4	3	4	0	0	0	1
15	2	1	6	1	0	1	1	1	1	1	0	2	1	2	2	0	1	1	1	3	0	0	0	1
26	1	2	4	3	1	3	1	2	6	3	0	3	5	4	1	0	2	6	4	4	0	0	0	1
29	0	4	4	3	1	3	5	1	6	2	0	3	4	5	0	0	2	6	3	3	0	1	1	0
16	2	1	3	1	0	1	3	0	3	2	0	1	3	1	0	0	2	5	3	1	0	0	1	0
22	3	3	4	2	0	0	0	3	4	3	0	2	2	4	3	0	2	3	2	3	0	0	0	1
22	2	2	9	0	0	2	1	2	3	1	0	1	2	3	2	0	2	4	2	2	0	1	1	0
15	1	0	4	2	0	0	3	0	4	1	0	2	2	0	1	0	2	3	2	3	0	0	1	0
55	0	8	8	3	2	0	7	8	15	4	0	2	12	5	0	0	2	7	8	6	1	0	1	0
28	1	3	5	3	1	0	5	3	4	3	0	3	4	5	0	0	2	4	4	2	0	1	1	0
34	1	4	9	3	0	2	4	4	6	1	0	3	4	5	1	0	2	5	5	4	0	0	1	0
20	2	2	4	2	0	0	3	2	2	3	0	1	1	2	1	0	2	1	1	2	0	0	1	0
33	6	4	5	0	1	0	2	4	8	3	0	1	6	3	3	0	2	7	4	4	0	0	1	0
52	1	8	9	1	4	2	8	8	8	3	0	2	6	8	1	0	2	6	5	6	1	1	1	0
25	6	3	5	1	1	1	0	3	4	1	0	3	2	3	3	0	2	3	2	3	0	0	0	1
20	1	0	6	0	0	2	1	0	9	1	0	1	7	0	1	0	2	6	5	0	0	1	0	0
20	4	0	4	1	1	2	0	0	5	3	0	3	4	0	2	0	2	5	4	1	0	1	0	0
25	2	1	5	1	0	0	4	3	5	4	0	1	4	4	2	0	2	5	4	2	0	1	1	0
25	2	1	10	1	0	0	0	2	6	3	0	1	5	2	1	0	2	5	4	2	0	0	0	1
15	1	0	7	0	1	1	0	0	5	0	0	2	4	0	1	0	2	5	5	3	0	1	1	0
30	1	1	9	4	1	2	4	2	4	2	0	2	3	2	0	0	2	6	3	5	0	0	0	1
35	2	0	6	5	1	3	5	2	8	3	0	3	4	2	1	0	2	5	4	2	0	0	1	0
18	2	0	3	2	2	1	2	0	3	3	0	2	2	0	2	0	2	3	2	3	0	0	0	1
20	1	1	2	2	1	0	4	2	5	2	0	2	4	2	1	0	2	4	5	3	0	0	1	0
15	0	1	2	1	1	0	3	1	3	3	0	2	1	2	0	0	2	2	1	3	0	0	0	1
46	2	2	9	6	1	6	3	2	12	3	0	3	6	3	2	0	2	9	5	6	0	1	1	0
38	1	4	8	3	1	3	0	4	11	3	0	3	7	4	1	0	2	6	7	4	0	0	1	0
14	0	1	1	1	0	1	3	1	4	2	0	1	3	1	0	0	2	3	3	1	0	0	0	1
17	1	0	3	2	2	2	0	0	6	1	0	3	2	0	1	0	2	4	2	2	0	0	0	1
58	1	6	8	6	2	6	5	6	12	6	0	3	8	5	1	0	2	7	6	5	0	1	1	0
14	1	1	4	0	0	0	4	1	1	2	0	0	1	1	1	0	2	2	1	1	0	0	1	0
20	0	1	2	3	1	3	2	1	5	2	0	2	3	1	0	0	2	4	3	3	0	0	0	1
24	2	1	6	1	2	1	4	1	4	2	0	1	2	2	1	0	2	1	2	5	0	0	1	0
12	0	0	6	0	0	0	1	0	3	2	0	0	1	0	0	0	2	2	1	3	0	0	0	1
24	1	3	6	1	2	1	2	3	3	2	0	3	1	3	0	0	2	2	1	5	0	0	0	1

20	0	0	3	0	0	3	3	0	9	2	0	3	5	0	0	0	2	9	6	2	0	1	1	0
20	0	0	6	1	3	1	0	1	4	4	0	3	3	1	0	0	2	5	3	5	0	0	0	1
25	2	1	5	0	2	2	4	2	4	3	0	2	4	3	2	0	2	6	4	5	0	1	1	0
25	2	1	4	1	2	1	3	0	9	2	0	3	4	1	1	0	2	6	5	6	0	1	1	0
30	2	2	9	3	2	1	2	0	7	2	0	2	6	2	0	0	2	6	3	4	0	1	0	0
15	1	0	2	1	0	2	3	0	4	2	0	2	2	0	1	0	2	3	2	2	0	1	0	0
20	1	3	4	1	0	0	0	1	6	4	0	1	4	3	0	0	2	7	4	4	0	0	0	1
25	2	0	6	4	1	2	1	0	7	2	0	2	5	0	1	0	2	6	5	3	0	1	0	0
30	1	1	7	2	2	3	3	0	6	5	0	3	5	1	0	0	2	9	6	4	0	1	1	0
40	3	3	12	0	1	2	6	3	8	2	0	2	5	5	2	0	2	8	5	5	0	1	1	0